Decision Support for Investment of Developer Effort in Code Review

by
Ruiyin Wen

Department of Electrical and Computer Engineering
McGill University

August 2018

A thesis submitted to McGill University
in partial fulfillment of the requirements
of the degree of
Master of Engineering

Abstract

Modern software development is a highly collaborative endeavour. Developers work in teams with tens, if not hundreds, of people who are globally distributed. At the heart of developer collaboration lies the process of code review, where fellow developers critique code changes to provide feedback to the author. Unlike the rigid formal code inspection process, which includes in-person meetings, the modern variant of code review provides developers with a lightweight, tool supported, online collaboration environment, where code changes are discussed. However, the existence of Modern Code Review (MCR) tools does not guarantee a smooth collaborative process that generates more value than cost. Indeed, the investment of developer effort in code reviewing is a key software development cost that needs to be spent efficiently and effectively.

Intelligent MCR investment decisions need to be made at the level of organizations and individuals. Thus, in this thesis, we set out to support team and individual code reviewing investment decisions. First, to support decisions about the *content of code reviewing feedback*, we train and analyze topic models of 248,695 reviewer comments from one open source community and one proprietary organization. We observe that more context-specific, technical feedback is being raised as the studied organizations have aged and as the reviewers within those organizations accrue project-specific experience. These topic models can be used to track organizational and individual feedback trends, and whether those trends align with respect to organization and individual reviewing goals.

Next, we set out to support individual decisions about *which review requests require additional effort*. Since patches that impact mission-critical project deliverables or deliverables that cover a broad set of products should involve more reviewing investment than others, we propose BLIMP Tracer—an impact analysis tool that pinpoints which deliverables are affected by given code changes. To evaluate BLIMP Tracer, we deploy a prototype implementation of it at a large multinational software organization, and conduct a qualitative empirical study with the developers from that organization. We observe that BLIMP Tracer not only improves the speed and accuracy of identifying the set of deliverables that are impacted by a patch, but also helps the new members of the organization to better understand the project architecture.

Résumé

Le développement moderne de logiciels est un effort hautement collaboratif. Les développeurs travaillent en équipe avec des dizaines, voire des centaines, de personnes réparties dans le monde entier. Au cœur de la collaboration avec les développeurs se trouve le procédé de revue de code, où d'autres développeurs critiquent les changements apportés au code pour fournir des commentaires à l'auteur. Contrairement au processus rigide d'inspection de code formelle, qui inclut des réunions en personne, la variante moderne de l'examen de code fournit aux développeurs un environnement de collaboration en ligne, où les modifications de code sont discutées. Cependant, l'existence d'outils de Revue de Code Moderne (RCM) ne garantit pas un processus collaboratif fluide qui génère plus de bénéfices que de coûts. En effet, l'investissement de l'effort de développement dans la révision de code est un coût clé de développement de logiciel qui doit être dépensé efficacement.

Les décisions d'investissement intelligentes du RCM doivent être prises au niveau des organisations et des individus. Ainsi, dans cette thèse, nous avons décidé de soutenir des décisions individuelles et organisationnelles d'investissement de la revue de code. Tout d'abord, pour soutenir les décisions concernant *le contenu des commentaires dans la revue de code*, nous formons et analysons des modèles de sujets de 248 695 commentaires de réviseurs provenant d'une communauté open source et d'une organisation propriétaire. Nous observons que des commentaires techniques plus spécifiques au contexte sont soulevés au fur et à mesure que les organisations étudiées ont vieilli et que les évaluateurs au sein de ces organisations accumulent une expérience spécifique au projet. Ces modèles de sujets peuvent être utilisés pour suivre les tendances de la rétroaction organisationnelle et individuelle, et si ces tendances s'harmonisent avec les objectifs de la revue de code des l'organisation et l'individuels.

Ensuite, nous avons décidé de prendre en charge des décisions individuelles concernant les requêtes de revue de code *requièrent des efforts supplémentaires*. Puisque les correctifs qui impactent les livrables de projet critiques ou les livrables qui couvrent un large éventail de produits devraient impliquer plus d'investissements de révision que d'autres, nous proposons BLIMP Tracer—un outil d'analyse d'impact qui identifie les livrables affectés par des changements de code donnés. Pour évaluer BLIMP Tracer, nous déployons un prototype dans une grande entreprise de logiciels multinationale, et menons une étude empirique qualitative avec les développeurs de cette organisation. Nous observons que BLIMP Tracer améliore non seulement la rapidité et la précision de l'identification de l'ensemble des livrables impactés par un correctif, mais aide également les nouveaux membres de l'organisation à mieux comprendre l'architecture du projet.

Related Publications

Earlier version of the work in this thesis were submitted or published as listed below:

- *A Longitudinal Study of Code Reviewing Feedback* (Chapter 4). Ruiyin Wen and Shane McIntosh. Submitted to Springer Journal of Empirical Software Engineering (EMSE), 32 pages.
- *BLIMP Tracer: Integrating Impact Analysis with Code Review* (Chapter 5). Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. To appear in Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME 2018), 10 pages.

The following publication is not directly related to the content in this thesis, but was produced in parallel to the research performed for this thesis.

- *Forecasting the Duration of Incremental Build Jobs.* Qi Cao, Ruiyin Wen, and Shane McIntosh. In Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017), pp. 524–528.

Acknowledgements

This thesis would be incomplete if I don't thank all souls who have played an important role in my life during my Master's study. First and foremost, I would like to thank my advisor, Shane McIntosh. Your enthusiasm for research inspired me to start this journey, and your guidance and support make this thesis possible.

I have had an amazing time working at the Software Repository Excavation and Build Engineering Labs (The Software REBELs), and have the privilege to know so many young, bright, and committed researchers in software engineering. I would like extend my gratitude to thank all Rebels: Keheliya Gallaba, Toshiki Hirao, Christophe Rezk, and Farida El Zatany for your camaraderie. Also, I appreciate Mathieu Nassif for proofreading the French version of the abstract. *Merci.*

My sincere thanks to my thesis examiner, Jin Guo, for her insightful feedback.

I appreciate Mitacs and Dell EMC for providing me an invaluable opportunity to complete a research project in industry. I feel lucky to have worked with all talented colleagues at Dell EMC. Special thanks to David Gilbert and Michael G. Roche for sharing their thoughts and helping me navigate the system.

A big shout-out to all my friends in Canada and abroad for the fun you brought me. Without you, the past two years would have been less colourful than it was.

Finally, I would like to thank my parents and my sister who encourage and support me at all costs. Also, to Jie Ni, thanks for your commitment and love. I am forever indebted.

To my buddy Frank Yang Zheng (1995–2015), who once dreamt of becoming a researcher.

Contents

Listing of tables

Listing of figures

1

Introduction

Modern software development is more than a personal struggle. It is not uncommon to have hundreds of developers working together on one complex software system. For example, 1,681 active developers from 225 companies collaborated to develop version 4.13 of Linux Kernel in 2017 [23]. Each individual developer is unlikely to understand every aspect of the complex software system. Hence, developer collaboration is necessary and challenging.

Code review is a mechanism that enables and enriches collaboration. Developers first write patches that implement new features or fix defects, then ask relevant reviewers to provide feedback or approve their patches. Code patches written by individual developers are peer-reviewed before integrating with the software system, which ensures a quality collaboration among a large base of developers.

Unlike the rigid code inspections of the past [27], the modern variant of code review uses tools to facilitate the need of a lightweight, online developer collaboration environment. Because of the light overhead, Modern Code Review (MCR) has been widely adopted by proprietary and open-source organizations

that have large teams for software development.

However, the mere existence of MCR does not guarantee a well-managed developer collaboration process. To truly improve the quality of a patch, reviewers must consider the potential implications of the patch and engage in a discussion with the author. Prior work shows that a lack of reviewer participation is correlated with a drop in software release quality [49, 70] and a drop in design quality [52]. Indeed, lax code reviews may allow poor quality code to slip through to official software releases, affecting customer-visible software quality negatively in large systems [48].

1.1 Problem Statement

Although MCR tools provide easy-to-use platforms for teams to collaborate in developing large scale software systems, they lack information that would help stakeholders to make pragmatic decisions about where to invest their time and effort.

Thesis statement: *Data about the content of past code reviews and the impact that a patch has on a software system can help stakeholders to make more effective effort-allocation decisions.*

To ensure a quality code review process, developers and managers should invest their resources efficiently and effectively. Bosu and Carver [16] find that developers spend an average of six hours per week reviewing code. Since spending equal amount of time to review each patch is not optimal, a busy developer should use available information to decide the priority of her backlogged patches for reviewing. From a team or organizational perspective, code reviewers and managers need to know how code reviews are being performed so that they can assess progress with respect to personal and community goals, and even guide future decisions. Since current code review tools do not provide the necessary information to support these use cases, in this thesis, we set out to provide frameworks for supporting (1) team awareness of code reviewing focus, and (2) individual decisions about which review requests require additional effort.

1.2 Thesis Overview

We now provide a brief overview of the thesis. Figure 1.1 provides an overview of the scope of this thesis. First, we provide the necessary background for our topic.

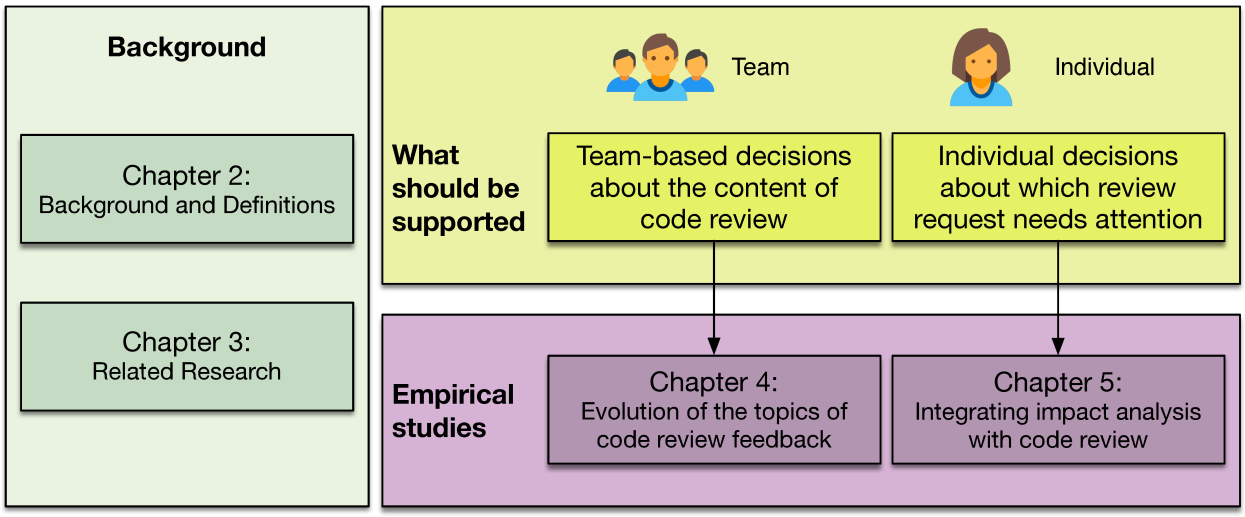


Figure 1.1: An overview of the scope of this thesis.

Chapter 2: *Background and Definitions*

Before discussing how to support stakeholders in allocating code reviewing resources, we first provide readers with background information and definitions of terms that we use throughout this thesis.

Chapter 3: *Related Work*

To situate this thesis with prior research, we present a survey of research on understanding and improving code review practices.

Next, we shift our focus to the main part of this thesis. In this thesis, we focus on concerns about how effort has been spent on code reviews of the past (team resource investment) and where effort should be spent on code reviews in a reviewing backlog (personal resource investment). We answer each concern separately using two empirical studies. Each empirical study is presented in its own chapter, as explained in the subsection below.

1.2.1 Team Resource Investment

Understanding how reviewing feedback evolves may help software project teams to focus on their collective reviewing goals. Review discussions are a rich source of information about the evolution of the system under review. Bacchelli and Bird [6] found that motivations for code review are both technical

(e.g., catching defects early) and non-technical (e.g., promote knowledge transfer). Rigby and Bird [61] found that the focus of code review has shifted from being on defect hunting to collaborative problem solving. Indeed, recent work has reported that roughly 75% of the issues that are uncovered [45] and fixed [11] during code review do not alter system behaviour, instead aiming to improve system maintainability. However, little is known about how reviewing feedback—a primary value-generating artifact of the code review process—changes as a software community and its stakeholders mature. In this thesis, we perform an empirical study that focuses on the evolution of code review feedback topics.

Chapter 4: *Topics of Code Reviewing Feedback*

While recent research analyzes the problems that are raised and fixed during code review, little is known about how reviewing feedback evolves as a software community ages and as reviewers accrue experience. To help teams make resource investment decisions based on the past trends of code review, we conduct a longitudinal study of 39,249 reviews that contain 248,695 review comments from a proprietary project that is developed by Dell EMC and the OpenStack Nova open source project.

1.2.2 Personal Resource Investment

Rigorous code review introduces an overhead on developers, whose time is a limited and valuable commodity. The time spent reviewing code is an expensive context switch away from other important development tasks (e.g. repairing and improving code). Making matters worse, patch authors at Microsoft report that an average of 35% of code review comments are not useful [17], suggesting that a large proportion of reviewing time may be misspent generating feedback that is not valuable.

Since some changes are of greater risk than others, some patches will require a more rigorous review than others. Czerwonka et al. [25] argue that spending an equal amount of reviewing effort on all code patches is a suboptimal use of development resources. Currently, to reduce waste in the reviewing process, developers use their intuition and their past experience to decide which patches require detailed feedback. However, knowing which patches require more reviewing attention than others is a difficult problem for code authors and reviewers alike.

Chapter 5: *BLIMP Tracer: Build Impact Analysis for Code Review*

To help reviewers make pragmatic decisions about where to invest reviewing effort, we developed Build Impact (BLIMP) Tracer, an impact analysis system that we integrated with the code review platform at Dell EMC. Unlike traditional change impact analysis [5], BLIMP Tracer operates on a Build Dependency Graph (BDG) that describes how each file in the system is processed to produce the set of intermediate and output deliverables. To evaluate BLIMP Tracer, we conduct a qualitative study with Dell EMC developers. We solicit feedback from participants during their use of BLIMP Tracer, and compare it with their current style of conducting impact analysis on patches.

1.3 Thesis Contributions

This thesis shows that:

- The change of reviewing behaviour of a code review community often coincides with project events (Chapter 4).
- Experienced reviewers who mature in different code reviewing communities focus on different topics according to project needs (Chapter 4).
- BLIMP Tracer not only made build impact analysis on code patches faster, but also vastly improves the depth and breath of impact analysis when compared to traditional methods (Chapter 5).
- BLIMP Tracer can also help to onboard new developers by helping them to better understand the system architecture (Chapter 5).

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background knowledge and definitions of key terms. Chapter 3 surveys research related to allocation of resources in code reviewing environments. Chapter 4 presents the result of an longitudinal study that reveals trends of code reviewing

feedback that help teams make collective investment decisions. Chapter 5 presents the design, deployment, and evaluation of BLIMP Tracer, our proposed impact analysis tool that integrates with code review platforms. Finally, Chapter 6 draws conclusion and discusses paths for future work.

2

Background

We use the term *code review* to refer to the activity where peer developers review patches and provide feedback to the author. In this chapter, we describe an overview of the code review process. Since we analyze and improve code review systems from the OpenStack and Dell EMC code review repositories, we focus on describing the process of their corresponding code review processes, which are enabled by *Gerrit*[*] and *Review Board*.[†]

2.1 The Code Review Process

Code review is a key software quality assurance practice, where fellow developers (reviewers) inspect changes to a codebase and provide feedback to the author. The broadly adopted contemporary variant of code review is tool-based and tightly integrates with the software contribution management process [12].

[*] https://www.gerritcodereview.com/
[†] https://www.reviewboard.org/

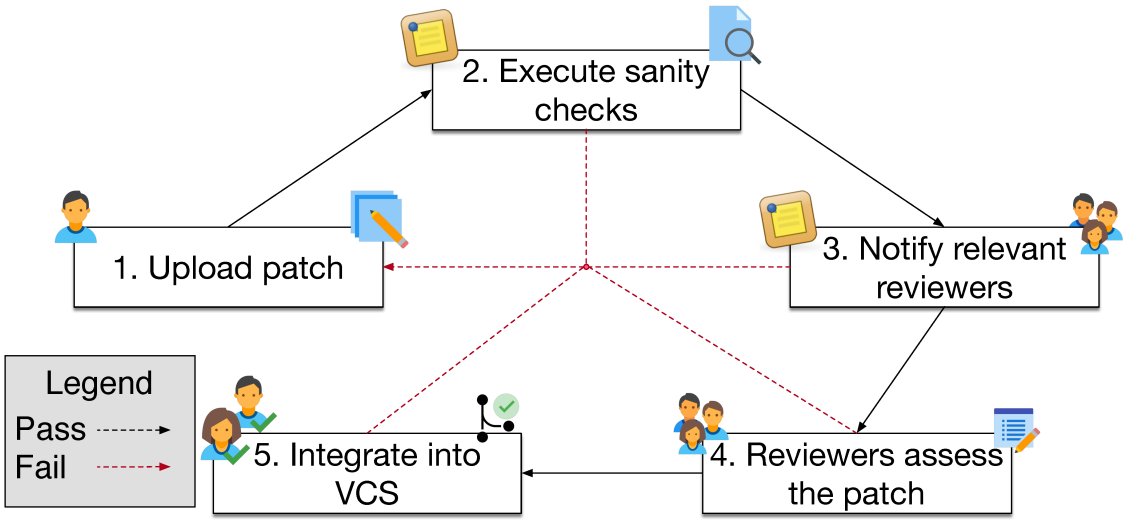


Figure 2.1: An illustration of the code reviewing process.

```
113     def handler(signum, frame):
114 »       raise NameError('Timeout Error')
```

Reviewer X:	I think you misunderstood me before, I meant it would be NameError...	Mar 24 3:13 PM
Change author Y:	def handler(signum, frame): raise Exception(TimeOutError()) def Tim..	Mar 26 4:15 AM

```
115
116     def setTimer(self,handler,execute,_indicate_start):
117     »   def wrapperTimer():
118 »       »       # Register the signal function handler
119 »       »       signal.signal(signal.SIGALRM, handler)
120 »       »       # Defined a timeout
121 »       »       signal.alarm(600)
122 »       »       try:
```

Reviewer X:	please use spaces instead of tabs	Mar 24 3:13 PM

```
123 »       »       »       self.execute() #  execute function called
124 »       »       except NameError:
125 »       »       »       self._indicate_start() # to reset in case of any issue
126 »       return wrapperTimer()
```

Figure 2.2: An example of inline comments in the Gerrit platform.

In order to provide context to this overview, we describe the code review process based on OpenStack and Dell EMC. A similar process is employed at most code reviewing tools elsewhere.

The OpenStack community uses Gerrit, and Dell EMC uses Review Board. Both tools are popular web-based code review platforms that tightly integrate with version control systems. As depicted in Figure 2.1, the code review process is comprised of five steps. We describe each step below.

1. *The author uploads changes to the code review platforms.*

   Once an author finishes changing the code, she uploads her set of changes to the code reviewing system. For Dell EMC developers, during the upload to Review Board, reviewers are recommended

based on the areas of the code base that were modified. However, Review Board prompts the author to confirm or replace those reviewers.

2. *The servers perform automatic verification as sanity check.*

   As a part of Continuous Integration (CI), the servers of the OpenStack community and Dell EMC initiate automated verification of the change to check for blatant mistakes (e.g., the codebase no longer compiles when the change is applied or incorrect formatting) before reviewers waste their time checking changes that are not ready for feedback. Normally, only when the code passes the automatic checks will reviewers begin their inspections.

3. *Relevant reviewers are notified.*

   The selected reviewers receive review requests, and should they accept, can begin their reviews of the patch. At the same time, they can decide to invite other developers to participate in the code reviewing process.

4. *The reviewers inspect the changes and initiate discussion.*

   Reviewers inspect the changes with respect to the previous versions of the system and provide feedback to the author. The Gerrit and Review Board web interfaces are designed to encourage reviewers to provide *inline comments*, i.e., comments that correspond directly to lines within the change. Figure 2.2 provides an example of two inline review comments. Reviewers may also write general discussion comments (non-inline) that summarize the inline comments, and provide further justification for their opinions, or further comment on the general content or form of the patch.

   In Gerrit, once reviewers finish with writing comments, they may provide a review score from -2 to +2 to indicate support for acceptance (positive values), support for rejection (negative values), or abstention (zero). On the other hand, core Review Board reviewers can assign 'Ship it!' labels to the patches that they deem ready to be integrated. The author may discuss with the reviewers by either replying to the inline comments or replying in the general discussion thread. If the code is not accepted for integration (e.g., due to insufficient support from reviewers or automatic verification failure), the author will need to improve the change by addressing the raised issues. After updating

the change, the author will upload a new revision of the change to Gerrit for another round of review.

5. *Integrate the approved code.*

   If the patch receives +2 scores (Gerrit) or 'Ship it!' labels (Review Board) from two members of the core team, it will be approved for integration into the main project repository. If any of the steps 2–4 fail, the patch returns to the author, who may revise the patch by addressing the feedback, and then the process repeats from step 2 onward.

2.2 Chapter Summary

This chapter provides some background knowledge of the code reviewing process. More specifically, we introduce the two largely similar code review processes that are used by the OpenStack community and the Dell EMC organization. In our experience, the process of code review is largely similar in other installations (e.g. Qt [48] and Sony Mobile [67]), setting aside some minor customization details.

In the next chapter, we survey prior research on understanding and improving code review in order to situate our empirical studies of investment of resources in code review with respect to the broader body of knowledge.

3

Related Work

In this chapter, we survey the related work on analyzing and improving code review. In addition, we include past studies that motivate us to use specific techniques in each of our two empirical studies.

3.1 Team Resource Investment

Past data from code review systems are needed to help teams manage their reviewing effort investment. The proliferation of code review data, and tools for analyzing it, have made several recent studies possible. Several papers have shared data sets of (and tools for interfacing with) Gerrit repositories [53, 32, 83]. Since these data sets can be quite large and difficult to understand, tools like ReDA [76] and Bicho [30] aim to support analysts in mining review data. In the same spirit of openness, we have made all the relevant data retrieved from the open-source project, OpenStack Nova, available online.

Code review is more than an exercise in defect hunting. Code review also serves as a platform for knowledge transfer, and collaborative problem solving [6]. Rigby and Storey [63] analyzed interactions in code

review processes of five open source systems and found that in addition to defect prevention, developers also talk about features, scope, and process issues. Baysal et al. [10] performed an empirical study on WebKit, and found the developer's affiliated organization and level of participation influence the outcome of code review. Jiang et al. [36] analyzed eight years of patch review data from the Linux kernel mailing list, and found that while the reviewing time becomes shorter, the time for integration becomes longer. Tsay et al. [77] analyzed GitHub pull requests and showed that although some third-party developers' patches are rejected after discussion, core developers often extract the ideas from the discussion and re-implement the rejected patches by themselves. Mäntylä and Lassenius [45] and Beller et al. [11] found that review discussions raise and fix three maintainability issues for every functional issue. Similar to prior work, in Chapter 4, we also analyze the rich data that is stored in code reviewing archives; however, we set out to better understand how the reviewing feedback that is generated changes as communities and reviewers mature.

Reviewer experience is a crucial factor that affects the value that is derived from review comments. Bacchelli and Bird [6] analyzed code review comments at Microsoft, and report that more a priori knowledge of the code triggers more valuable feedback. Bosu et al. [17] analyzed the usefulness of 1.5 million review comments in Microsoft, and found that reviewers make more useful comments when they have worked for Microsoft for a longer period of time. Rigby et al. [62] examined 25 open source software projects and found that people with more expertise in code review are the ones who provide context-specific feedback. Di Biase et al. [26] manually analyzed 185 security issues by backtracking them to the code review stage. They found that reviews that were conducted by two or more reviewers tend to be more successful at finding security issues. Since the prior work demonstrates that reviewing expertise is a key skill, in Chapter 4, we study how reviewing feedback changes as reviewers accrue experience.

To ensure that reviewers who have the right expertise are invited to review, recent work proposes approaches to recommend reviewers. These approaches derive recommendations using the history of the code [8, 75, 72], reviewer profiles [84, 59], and the textual content of patches [81]. In Chapter 4, instead of recommending reviewers, we advocate for the use of topic models to track trends in community and reviewer activity. The eventual goal is to help teams better manage their focus on the collective goals on

their projects.

Recently, there have been studies on different types of code review comments. Pangsakulyanont et al. [56] used semantic similarity to group 72,000 review comments into different topics, and found that code review comments are often unrelated to defect prevention, with some of them discussing trivial issues. Kononenko et al. [40] used the SZZ algorithm [68] to detect that 54% of the reviewed changes in Mozilla are bug inducing, concluding that code reviewers tend to miss bugs. Kononenko et al. [39] also surveyed 88 core Mozilla developers and found that review quality is mainly associated with the thoroughness of the feedback. Moreover, reviewers find it difficult to maintain their technical skillset for writing high-quality reviews. Norikane et al. [54] claimed that different kinds of code review feedback affect the willingness of a volunteer contributor to engage in open source software. Zhu et al. [87] showed that improving code review management, e.g., providing clearer guidelines for reviewers, will make code contribution more efficient in software projects. We believe that topic models are a viable approach to support a community and reviewer analytics dashboard that would enable a data-driven approach to code review management.

3.1.1 Topic Modelling

Topic modelling has been used in various experiments that appear in the software engineering literature in the recent years. Xia et al. [82] used topic modelling to automatically recommend tags to describe the most important features of posted content or projects. Zhao et al. [86] used LDA to extract topics from discussions involving bug fixes in five open source projects, and explore the relation between the frequency of discussion and bug reworking. Maskeri et al. [46] used LDA to extract business topics from identifiers and comments in source code. In Chapter 4, we use LDA to extract and identify topics from code review comments.

In addition to identifying topics from a text corpus, previous work has also focused on the analysis of the trends that are shown in topics. Barua et al. [9] used LDA to extract topics from StackOverflow, a popular Q&A forum, and identified the changes of technical topics and programming language over time. Linstead et al. [43] applied LDA on source code and investigate the evolution of programming

concepts from smaller code bases to larger ones. Hindle et al. [34] applied topic models on developer communication corpora within pre-defined time windows, and visualize the topics and their trends over time according to those windows. We focus on the changes of topic choices with respect to time and developer experience.

Since topic modelling is widely used in software engineering research, previous literature has pointed out issues and suggested different ways to tackle them. Chen et al. [22] surveyed 167 software engineering papers that use topic modelling techniques for different purposes. They pointed out common pitfalls, such as issues in interpretation and parameter settings. They suggest that software engineering researchers should keep up with the machine learning community to avoid well-known pitfalls while applying topic modelling techniques. Hindle et al. [33] surveyed developers and project managers, asking them how they interpret topics that were generated by topic models. They found that the level of difficulty for interpretation differs across topics. We recognize the importance of topic model construction and validation by using work from the natural language processing community [21, 85] to support us when training our models.

3.2 Personal Resource Investment

Understanding the architecture of the software system is crucial as software defects may often be related to incorrect dependencies. Seo et al. [66] studied 26.6 million builds at Google and observe that most of the build failures are associated with dependencies (i.e., design or architectural-level component interactions). Indeed, Paixao et al. [55] revealed that developers are generally not aware of architectural changes. They analyzed code review data from four open source systems in conjunction with their commits, and found that only 38% of time do developers discuss the impact of their changes on the architectural structure. To aid in exposing developers to the higher level impact of their changes, in Chapter 5, we propose BLIMP Tracer, a build impact analysis tool that plugs into the code reviewing interface. The long term vision of BLIMP Tracer is to improve software quality by more clearly explaining to developers what the impact of their patches are. Armed with that clearer understanding, reviewers and testers can focus their effort more effectively.

3.2.1 Build Impact Analysis

We derive the definition of build impact analysis from that of change impact analysis. Change Impact Analysis (CIA) refers to the efforts to identify the potential consequences of a change to a software system [5]. Similarly, build impact analysis finds the consequence of a change with regard to build system inputs (source code, data files) and outputs (project deliverables, products).

Researchers have explored ways to conduct CIA for software in different languages and at various granularity levels. Ren et al. [60] designed *Chianti*, which uses the interdependent changes' history to determine change impact for Java programs. Apiwattanapong et al. [4] introduced an algorithm that uses a small amount of dynamic information to efficiently analyze change impact at the level of methods. Gyori et al. [31] proposed an algorithm that uses equivalence relations to discover change impact at the level of statements. Li et al. [42] surveyed 30 academic publications and found that although CIA is increasingly crucial in software maintenance, most of the proposed tools in academia are yet to be applied in industry. To bridge the gap, in Chapter 5, we describe an impact analysis tool (BLIMP Tracer) that we developed and integrated with a production code reviewing environment in industry.

Researchers have proposed techniques to analyze data in previous studies with respect to impact analysis and build system analysis. Breech et al. [18] used static analysis to estimate the influence of a change by considering scoping, function signatures, and global variable accesses. Canfora and Cerulo [19] used information retrieval algorithms to link the text-based change request description and the code entities impacted by the change. Jashki et al. [35] proposed an impact analysis technique that creates clusters of closely associated files by mining their co-modification history in version control systems. Tamrawi et al. [69] proposed SYMake, an infrastructure and tool that evaluates Makefiles symbolically, and used it to detect code smells and errors. Al-Kofahi et al. [3] developed MkDiff to detect changes to a Makefile at the semantic level. Adams et al. [1] designed MAKAO, a tool for visualizing, querying, refactoring, and validating build dependency graphs through parsing build logs. BLIMP Tracer combines impact and build system analyses by providing build impact analysis report based on information retrieved from build data in the past.

3.3 Chapter Summary

In this chapter, we survey prior research with respect to team and personal code reviewing resource investment—the two key themes of this thesis. We find that although researchers have analyzed code reviewing behaviour to make teams and developers' reviewing effort investment more efficient, they have yet to incorporate natural language processing or impact analysis techniques to do so.

In the following chapters, we describe our empirical studies that use these two techniques to bridge the gaps in the literature. We begin, in the next chapter, by studying the evolution of code reviewing feedback, and how it can be used to support team decisions about where to invest code reviewing effort.

An earlier version of the work in this chapter has been submitted to the Springer Journal of Empirical Software Engineering (EMSE).

4

Evolution of Code Reviewing Feedback

4.1 Introduction

Code review is a process whereby fellow developers inspect code changes and provide feedback to the author. Code review is a valuable mechanism that software teams use to improve software quality. Recently, software teams have begun to adopt tools that are developed specifically for managing the code review process. These tools support remote, online code reviews, storing the generated data in a code review database.

Unlike the rigid code inspections of the past [27], the modern variant of the code review process is informal; however, review discussions are still a rich source of information about the evolution of the system under review. Bacchelli and Bird [6] found that motivations for code review are both technical (e.g., catching defects early) and non-technical (e.g., promote knowledge transfer). Rigby and Bird [61] found that the focus of code review has shifted from being on defect hunting to collaborative problem solving.

Indeed, recent work has reported that roughly 75% of the issues that are uncovered [45] and fixed [11] during code review do not alter system behaviour, instead aiming to improve system maintainability.

The value that is derived from a code review process is dependent on the investment that reviewers make when reviewing. For example, McIntosh et al. [48] found that the mere existence of a code review (i.e., code review coverage) shares a weaker link with post-release software quality than measures of review participation do. Thongtanunam et al. [74] found that reviews with poor participation can be explained using characteristics of the change, its author, and its reviewers.

Little is known about how reviewing feedback—a primary value-generating artifact of the code review process—changes as a software community and its stakeholders mature. Understanding how reviewing feedback evolves may help software project teams to reduce the ramp-up time of newcomers to their communities.

In this chapter, we perform a longitudinal study of code review feedback in two large, rapidly evolving software communities. Our data set is comprised of a corpus of 248,695 reviewer comments from 39,249 changes that we extract from two projects that are developed by the Dell EMC (proprietary) and OpenStack (open source) communities. We train and analyze topic models using Latent Dirichlet Allocation (LDA) for each of the studied projects. These models show that context-specific, technical issues (e.g., configuration and API-related topics) are more frequently discussed than formatting issues (e.g., whitespace and spelling errors) overall in both communities. We then use the topic models to perform a longitudinal study, which addresses the following two research questions:

RQ1. How does the popularity of code review topics change as a community ages?

Motivation: Developers discuss different topics in different phases of the evolution of a project. Prior work [63, 77] has analyzed the content of code review discussions. However, little is known about how this content changes as a software community matures. For example, after introspection, a community may adjust code review focus to address its perceived shortcomings. Alternatively, as time passes, a community may tacitly degrade in its code reviewing focus. Thus, we are interested in how the content of review discussions change as a software community matures.

Results: Our topic modelling approach can reveal interesting trends in the reviewing behaviour

of a community. Topics related to *exception handling* and *object oriented design* concerns have become significantly less prevalent as the Nova project has aged. Conversely, context-specific, technical feedback has become more prevalent in both studied projects. We also observe interesting changes of trends for *code review process*, *logging and user-facing error messages*, and *object-oriented design* topics in the Dell EMC project between late-2015 and early-2016, which coincides with a large change in team composition.

RQ2. How does the popularity of code review topics change as reviewers accrue experience?

Motivation: It is often expected that novice reviewers focus on generic flaws in the code (e.g., code style), while the more experienced reviewers produce reviews that are more context-specific. A software community is made up of developers who are (ideally) learning from their past experiences and improving. Prior work [17, 39, 62] has pointed out that the more experienced reviewers are often the authors of high-quality review comments. In this research question, we set out to gain a better understanding of how reviewers change their reviewing focus as they accrue experience.

Results: Our topic models also yield interesting trends in reviewing behaviour as reviewers accrue experience. The more experienced reviewers in both studied projects tend to focus more on context-specific, technical feedback, suggesting that their reviewing skills are honed to provide feedback with a greater return on investment as they accrue experience within the teams. In addition, we observe trends that coincide with team focus. For example, our Dell EMC topic models shows that as reviewer experience grows, so does the amount of *code style* feedback. This coincides with a concerted effort that senior Dell EMC developers have made to provide *code style* feedback to help to onboard a recent influx of new developers.

Our topic models lay the groundwork for an analytics system that could be used to monitor trends in reviewing focus. The proposed measures that we extract from topic models could be tracked with respect to community and/or personal goals.

Table 4.1: An overview of the studied projects.

Project	Scope	# Changes	# Comments	Time
OpenStack Nova	Provisioning management for OpenStack	26,547	154,171	2011.09–2018.01
Dell EMC Project	Enterprise data backup & recovery solution	12,702	94,524	2013.09–2017.09
Total	-	39,249	248,695	-

4.1.1 Chapter Organization

The remainder of this chapter is organized as follows. Section 4.2 describes the design of our case study. Section 4.3 discusses the overall prevalence of the extracted topics in the data set, while Section 4.4 presents the results of our longitudinal study with respect to our two research questions. Section 4.5 discusses the broader implications of our results. Section 4.6 describes the threats to the validity of our study. Finally, Section 4.7 draws conclusions.

4.2 Case Study Design

In this section, we describe the design of our case study. First, we provide our rationale for selecting the OpenStack Nova and Dell EMC projects as subjects for our study (Section 4.2.1). Next, we explain how we extract and preprocess the data (Section 4.2.2). Finally, we describe our approach to training the topic models that we use to address our research questions (Section 4.2.3).

4.2.1 Studied Communities

We choose to study projects in the OpenStack and Dell EMC communities because we want to perform a case study on large, widely-used and rapidly-evolving software with a globally distributed development community. Table 4.1 provides an overview of the studied projects. In total, we analyze 39,249 changes that contain 248,695 review comments.

As a cross-company open source community, OpenStack has a vested interest in improving their code review process. In the past, researchers have recognized the value of the data from the OpenStack com-

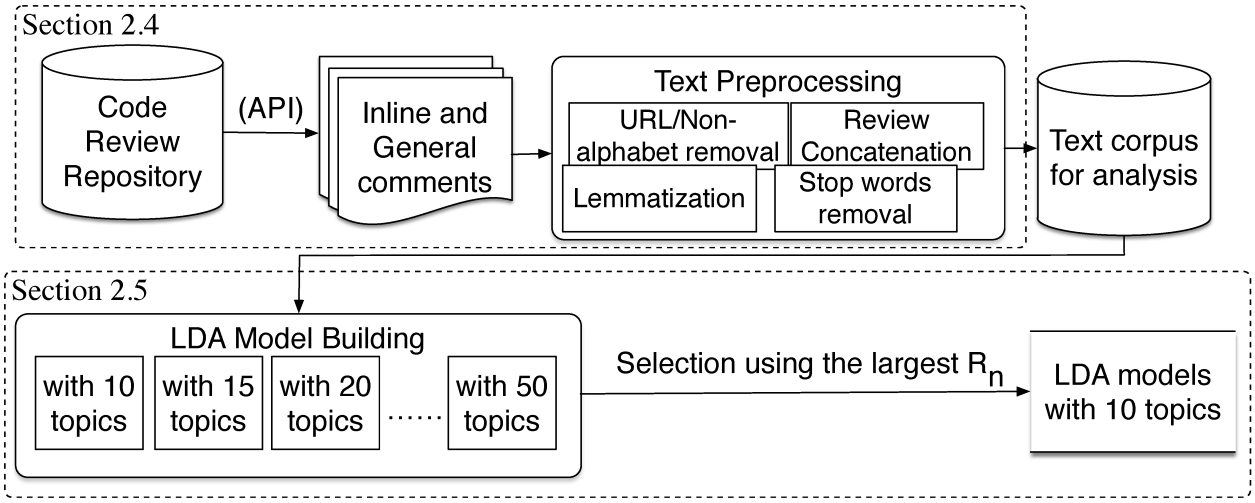


Figure 4.1: The overview of the data preparation and model training process.

munity, using it to evaluate reviewer recommendation approaches [75], analyze the relationship between reviewing and authoring expertise [73], evaluate the fairness of code reviews [28], study the career paths of developers [79], and study the evolution of company participation in open source development [29].

The OpenStack community develops software that manages large pools of compute, storage, and networking resources, and is used to support a wide array of business applications.* We choose to analyze Nova—the provisioning management system of OpenStack—because it attracts the most developers and reviewers when compared to the other OpenStack projects.

To combat the threat of generalizability, we also analyze the code review data of our industrial partner at Dell EMC. This Dell EMC project provides an enterprise-level solution that orchestrates data backup for disaster recovery.

4.2.2 Data Extraction and Preprocessing

Since both inline and general comments reflect reviewer critiques of changes, we decide to train our topic models using a text corpus that includes reviewer-produced inline and general comments. Our industrial partners at Dell EMC provided us with a replica of their live database of code reviews from which we can extract the code review comments. On the other hand, to extract the necessary data from the OpenStack Nova project, we download the corresponding comments for each proposed change using the Gerrit

*https://www.openstack.org/

API.[†]

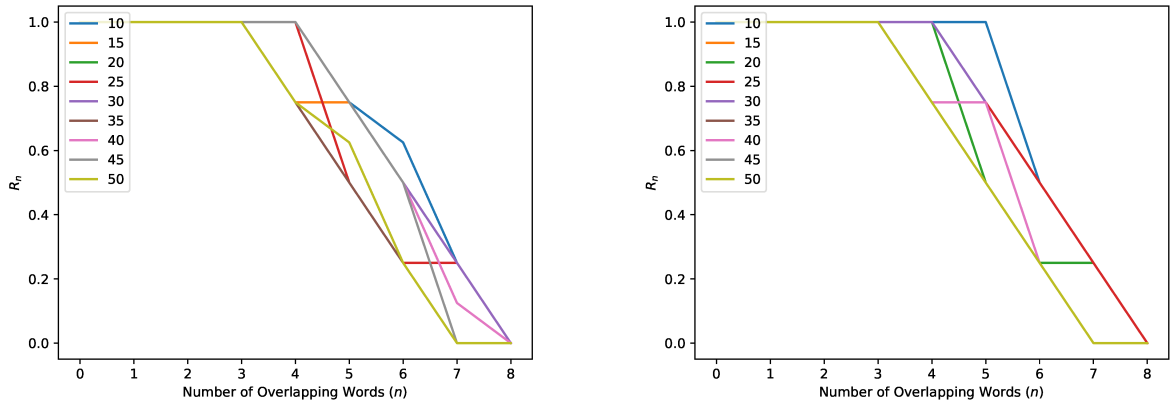
We further filter both sets of comments to remove those that were produced by bots (e.g., integration and testing bots) and replies that were written by the authors of the changes themselves. After applying these filters, our data set contains only the comments that were written by the reviewers of each change.

In order to mitigate the impact of noise on our topic models, we identify whether the comments are natural language or code by using `NLoN`, an R package that determines whether a document is natural language using machine learning algorithms [44]. After that, we apply standard text preprocessing techniques [37] to each document of inline comments. We first remove URLs, non-alphabet characters, and convert words to lower case. Then, we remove stop words—the most common English words that add little lexical meaning (e.g., the, a, be). We use the list of stop words from the RANKS NL Page Analyzer,[‡] a widely-used list in Search Engine Optimization (SEO), which includes 173 generic prepositions, pronouns, verbs, and adjectives. Finally, to minimize the effect of conjugation and synonyms, we apply lemmatization to each token of the comment corpus. Lemmatization maps different conjugated forms of a word to their base form according to its part-of-speech tag. We use lemmatization instead of stemming approaches (e.g., Porter Stemmer [58]), since lemmatization tends to preserve more of the original meaning of the word [37].

4.2.3 Topic Modelling

Topic models are a type of statistical model that discover latent topics in a corpus of text documents. In our research setting, our corpora are comprised of documents of comments that we extract from the code review databases of the OpenStack Nova and Dell EMC communities. We use Latent Dirichlet Allocation (LDA) [15] to automatically detect the latent topics in the preprocessed review comment corpus. Researchers have developed several topic modelling techniques for different goals [41, 71]. LDA meets our research goal, as it groups discussion topics in natural language text documents [15].

[†] https://review.openstack.org/Documentation/rest-api.html
[‡] https://www.ranks.nl/stopwords



(a) R_n of Nova.

(b) R_n of Dell EMC.

Figure 4.2: The R_n (*median number overlaps of size n words* in cross-run topic models with same K values) for both analyzed projects. Lines coloured by different K values.

Overview of LDA

LDA is probabilistic in nature, and provides multiple ways to assess a topic and its related words. As LDA is a statistical algorithm, it represents topics as probability distributions over the corpus, and each word in the corpus follows a probability distribution over a topic. LDA groups words into topics using their co-occurrence frequency in the corpus documents. As similar-meaning words tend to co-occur more frequently than different-meaning words, words within topics are often semantically related. Thus, LDA can associate frequently co-occurring words with higher level concepts (i.e., topics).

LDA Implementation

LDA uses machine learning algorithms that infer a topic membership distribution using the documents within an input corpus. In this chapter, we use the LDA implementation that is provided by MALLET [47]. MALLET provides an implementation of Gibbs sampling for training LDA models, and is widely used within the software engineering domain [9, 64, 38].

Choice of Parameters

Before training an LDA model, one must provide several parameters to initialize process, including the number of topics (K), the probability of topic t in document d ($\alpha = P(t|d)$), and the probability of word

w in topic t ($\beta = P(w|t)$). In the MALLET implementation of LDA, α and β can be initialized at random and automatically tuned via a re-sampling process; however, a K value must be provided a priori.

With a K value that is too large, topics may become fragmented and lose their semantic meaning. On the other hand, with a K value that is too small, topics may become tangled and contain more than a single semantic meaning. Indeed, selecting the correct value for K is an important step in the LDA modelling process, but is still an open research problem [22].

Agrawal et al. [2] suggest that since parameters widely affect the stability of topic models, they should be tuned independently for topic models built for different corpora. We tune the parameters in aiming to produce a topic model that has high stability, i.e. future researchers can easily reproduce a similar topic model using our data set. In order to acheive model stability, we first train models with $K = [10..50]$ five times with randomly initialized α and β values. Then, for each set of models with the same K value, we calculate the R_n, i.e., the median number overlaps of size n words. R_n is a metric that measures the cross-run similarity of topics [2]. More specifically, R_n is the median number of occurrences of n terms appearing in all the topics in all runs.

Figure 4.2 shows the R_n values for each set of topics trained using different K values. We observe that the R_n curves are the highest when $K = 10$ for both studied projects, i.e., the $K = 10$ set of topic models share the most similarity with each other, and are hence the most stable. Therefore, we use a configuration of $K = 10$ for the remainder of our analyses.

Output of LDA

Once trained on our preprocessed data, LDA outputs a set of topics that contain statistical distributions of words in the corpus. The words with higher probability scores often correspond to a high-level concept. For example, if the words with the highest probability scores in a topic are "log", "message", "error" and "debug", this indicates that (1) these words co-occur frequently in documents of the corpus and (2) they likely fit a similar high-level concept. In this case, we suspect that the topic is related to logging and exception handling, and label the topic as such.

LDA also generates a distribution of topic membership scores for any given document. More specif-

Table 4.2: A map between the topics' indices, their labels, and their topic share values.

Theme	OpenStack Nova	Share	Dell EMC Project	Share
Context Specific	Volumes and Storage Management	6.1	File Locations	4.5
	Provisioning Decision Making	4.4	Project Configuration	6.5
	Virtual Machine	5.1	Project Terminology	8.8
	API Issues	10.5		
Exception Handling	Exception Handling, Logging and User-facing Error Msg	7.3	Logging and User-Facing Error Msg	9.3
			Exception Handling and Memory Management	8.8
Language Specific	Python Collections	8.1	String/Buffer Issues	5.4
Design	Object Oriented Design	12.4	Object Oriented Design and Concurrency	10.4
			Function Design	11.9
Code Review Process	Code Review Process and Minor Issues	20	Code Review Process and Minor Issues	14.8
Code Style	Code Style	6.8	Code Style	8.2
Unit Testing	Unit Testing	6.2		

ically, for a given document d_i, the LDA model outputs a membership score $0 \leq \delta(d_i, t_k) \leq 1$, which indicates the strength of the relationship between d_i and topic t_k (larger values indicate a strong relationship). For example, the review comment "provide more straightforward error messages and log them appropriately" will have a strong relationship with the logging and exception handling topic described above, and thus, will have a high membership score for that topic.

4.3 Topic Prevalence

Prior work has analyzed the contents of code review comments. For example, Bacchelli and Bird [6] found that code reviews at Microsoft contain code improvement suggestions and requests for additional detail, in addition to addressing code defects. Mäntylä et al. [45] and Beller et al. [11] find that there are roughly three maintainability comments for every functionality comment in the code reviews of several proprietary and open source systems.

To complement these observations from the literature, prior to performing our longitudinal analysis,

we are interested in how prevalent our automatically extracted topics are in our data set.

4.3.1 Approach

We first identify the high-level concepts that the ten LDA-generated topics for both of the studied projects highlight. More specifically, we label them by reading the 20 terms and 20 unprocessed review comments with the strongest association to each topic.

We select the terms with the top 20 term weights for each topic. When ordering the terms of which a topic is comprised, we draw inspiration from the Term Frequency - Inverse Document Frequency (TF-IDF) concept. The TF score is mapped to the term weight within the topic. The IDF score is mapped to the Inverse Topic Frequency (ITF). We order terms by their TF-ITF score—terms with high term weight scores that appear in few other topics are considered first in our topic labelling process.

To further aid in the topic labelling process, we also analyze the 20 review comments with the strongest association to the topic under analysis. We only include comments with highest $\delta(d_i, t_k)$ scores for each topic t_k to avoid including comments for which the topic membership is less definitive. The first author manually labelled each topic and the second author confirmed the labels in follow-up meetings.

After labelling the topics, we apply the topic share metric of Barua et al. [9] to each topic t_k across the corpus of review threads. The topic share metric is defined as:

$$topic\_share(t_k) = \frac{1}{|D|} \sum_{\substack{\forall d_i, d_i \in D, \\ (d_i, t_k) \geq 0.1}} \delta(d_i, t_k) \tag{4.1}$$

where D is our corpus of review comments and d_i is an individual document (a review comment). The topic share measures the proportion of documents in the text corpus that contains a specific topic. For example, $topic\_share(t_1) = 0.28$ indicates that 28% of the documents share a non-negligible association with topic t_1.

Since the topic membership score δ is statistical in nature, each comment has a distribution of membership scores for all topics. Since we train topic models with $K = 10$ topics, each topic will have a minimum membership score of $\frac{1}{10} = 0.1$ for a (theoretical) document that is not associated with any topic. On the

other hand, if a document is associated with some topic t_i, the topic membership score for some other non-related topics must be less than 0.1. Therefore, to keep the topics that have the minimum membership score from skewing our topic share scores, we filter out topic membership scores below 0.1.

4.3.2 Topic Identification

As mentioned above, we analyze strongly associated terms and example threads when labelling our topics. Below, we provide a sample comment from two topics that share the same theme in both of the OpenStack Nova and Dell EMC projects. In addition to these comments (and 19 other similar comments for these topics), we analyze the top 20 terms of the topics in the studied systems (40 terms total).

> No, this is correct indentation for a continued line. Visual indentation would only be 4 spaces, which would line up with the 'return' line below. Correct indentation is 2 levels, or 8 spaces.
> *Project: Nova, topic_score(code style) = 0.94*

> Are you intentionally indenting this much space for a command that doesn't fit? Also, because all these are at the same level it makes it a bit harder to tell when commands begin and end. You may want to consider shifting all the sub-commands an indent to the right.
> *Project: Dell EMC, topic_score(code style) = 0.96*

We label these topics as *code style* because the sample comments focus on issues like indentation and visual appearance of the code, and the top 20 terms include keywords like "line", "space", and "blank".

We present the labels that both authors agree upon in Table 4.2. We include the full mapping from topics to their most relevant terms in Appendix A.1. The full mapping from topics to their most relevant comments in Nova are included in our online replication package[§].

4.3.3 Observations

In general, the proportion of review comments that are associated with context-specific topics (i.e., topics that tackle project-specific issues) is on par with code style or code review process discussions. Table 4.2

[§] https://github.com/software-rebels/code-review-topic-models

shows that the code style and code review process topics have a total topic share of 26.8% and 23% for Nova and Dell EMC, respectively. On the other hand, four topics in Nova and three topics in Dell EMC are context-specific, and have a combined topic share of 26.1% and 19.8%. As a result, approximately half of the comments belong to technical topics that are related to general software engineering concepts and are comparable across the studied projects.

These results complement observations of prior work. For example, Bacchelli and Bird [6] found that although general communication is an important aspect of code reviewing practice in Microsoft, code improvement topics are more frequently discussed. Moreover, our topic models yield a similar rate of *code style* issues (6.8%–8.2%) as was observed through manual review inspection by Mäntylä and Lassenius [45] in a different context (*visual representation* concerns are raised in 9.8%–10.8% of their studied reviews).

4.4 Case Study Results

In this section, we present the results of our case study with respect to our two research questions. For each research question, we first present our approach to addressing it (including the measures that we use to operationalize key concepts), followed by the results, and our observations.

RQ1: How does the popularity of code review topics change as a community ages?

RQ1: Approach

In order to analyze how topic popularity changes over time, we analyze trends using the *topic impact* measure [9], which defines the impact of a topic t_k in month m as:

$$topic\_impact(t_k, m) = \frac{1}{|D(m)|} \sum_{\substack{\forall d_i, d_i \in D(m), \\ (d_i, t_k) \geq 0.1}} \delta(d_i, t_k) \tag{4.2}$$

where $D(m)$ represents the set of review comments that are written in month m. In other words, topic impact measures the proportion of review comments that are associated with a given topic t_k in a given

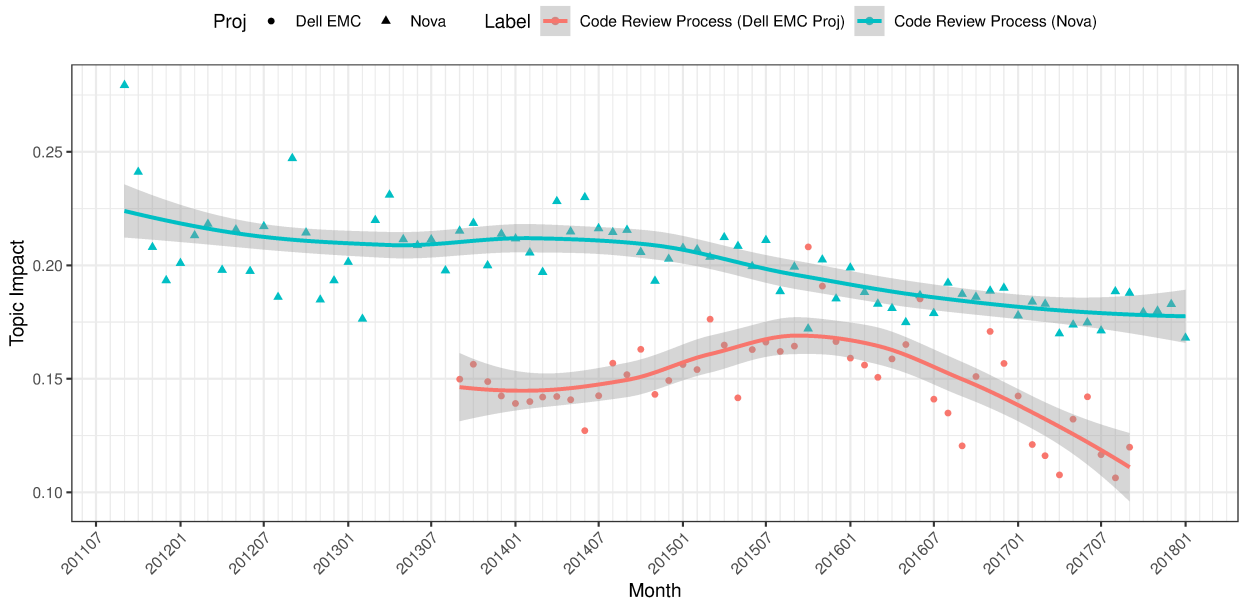


Figure 4.3: The impact score of topics discussing *Code Review Process* issues, plotted with regard to time.

month m. Similar to the topic share measure, we keep negligible membership scores from skewing topic impact values by filtering out topics that have membership scores below 0.1.¶

We first compute the topic impact score in all of the studied months for each topic. Then, we show key plots of the trends over time. For the purpose of cross-project topic comparison, we group the figures according to the concepts to which each topic belongs. We then analyze the trends of topics in both projects that belong to a similar concept. In addition to the raw values (scatterplot points), we plot a trend line using Loess-smoothed regression lines. The translucent grey shaded area shows the 95% confidence interval.

In order to determine if the impact of a topic is significantly increasing or decreasing over time, we use the Cox-Stuart trend test [24]. The Cox-Stuart trend test compares the earlier data points with later data points to check for significantly increasing/decreasing trends. We apply the two-tailed variant of the test to all of the topic impact trends using a 95% confidence level ($\alpha = 0.05$).

RQ1: Results

Observation 1 — As the studied communities mature, reviewers tend to provide fewer code review comments related to the theme of code review process. Code review process topics mainly include comments that address issues related to procedural formalities and minor issues of the code review process. Figure 4.3 shows a consistent decreasing trend of the Nova project. On the other hand, we observe that the trend of this topic in the Dell EMC project is non-monotonic, slightly increasing before late 2015 and decrease afterwards. To dig deeper into this trend, we held a discussion with a development manager from the Dell EMC project who explained that the period with the most steady growth (early/mid 2015) coincides with a large change in the composition of the development team. During this period, there was a large influx of new members into the development team. This may explain why more comments appear with respect to code review process in that period. As the new members became more familiar with the code review process, the topic trend begins to descend (early 2016).

Observation 2 — The Dell EMC project has less discussion within the code review process topic during the analyzed period. Despite the growing trend during 2015, Dell EMC developers seem to raise *code review process* concerns than those of Nova. Figure 4.3 shows that the Dell EMC trend never exceeds the Nova trend. There are several reasons why this may be occurring, but we suspect that offline discussion is a large factor. Although the teams themselves in the Dell EMC project is globally distributed, within the teams, most members are collocated. Thus, process-related discussions may occur in offline discussions more frequently than in the Nova project, where team members are spread across several software organizations.

Observation 3 — Although the code style topic in Dell EMC is relatively flat, Nova reviewers focus less on code style over time. Figure 4.4 shows that the *code style* topic in Nova has a decreasing trend, which a Cox-Stuart test confirms is significant. Compared to the Dell EMC project, which dates back to early 1990s, Nova is a relatively new open-source project. Our code review dataset for Nova includes the initial phases of its development. The larger initial quantities of *code style* comments may be an artifact of the

¶ A sensitivity analysis that explores threshold values between 0.1 and 0.2 yielded no significant change in the shape of the topic trends. See the online replication package for more detail.

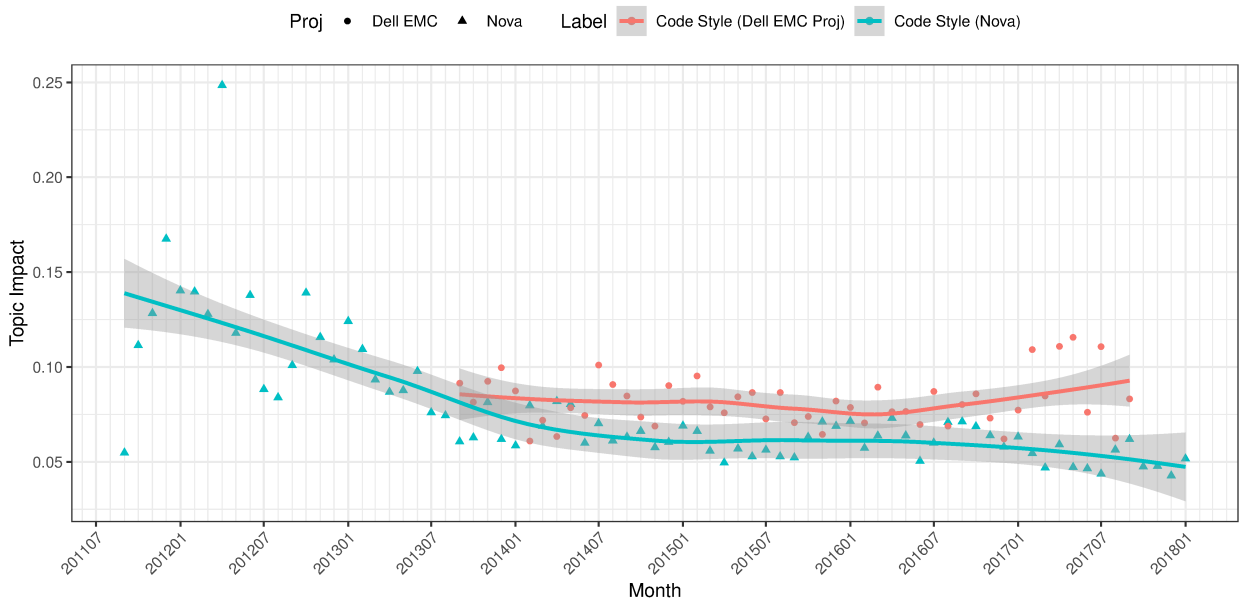


Figure 4.4: The impact score of topics discussing *Code Style* issues, plotted with regard to time.

early phases of Nova review adoption. Since the *code style* topic has a lower bar to entry for reviewers, those with little expertise can provide such feedback; however, *code style* comments may be less valuable to authors, who expect to receive technical feedback [6, 17]. The decreasing trend in *Code Style* as Nova has matured is a positive indication.

On the other hand, the Dell EMC project has a long history of performing code reviews, including code inspections and code walkthrough meetings, which were not recorded by any tool. Moreover, many code style checks are automated as part of the code check-in procedure, so it is not surprising that there is no significant trend in the *code style* topic for the Dell EMC project.

It is also interesting to note that Nova's trend for the *code style* topic has dropped slightly below that of the Dell EMC project. We suspect this may be due to a concerted effort of senior Dell EMC developers to improve code style practices. We discuss this further in RQ2.

Observation 4 — Although two out of three topics in exception handling in both projects show a decreasing trend over time, we find a change of trend for the logging and user-facing error message topic for Dell EMC in late 2015. There are two topics in the Dell EMC project and one in Nova that share the theme of *Exception Handling*. The Dell EMC project is primarily written in C. Hence, members in the Dell EMC community discuss exception handling issues related to memory management frequently enough

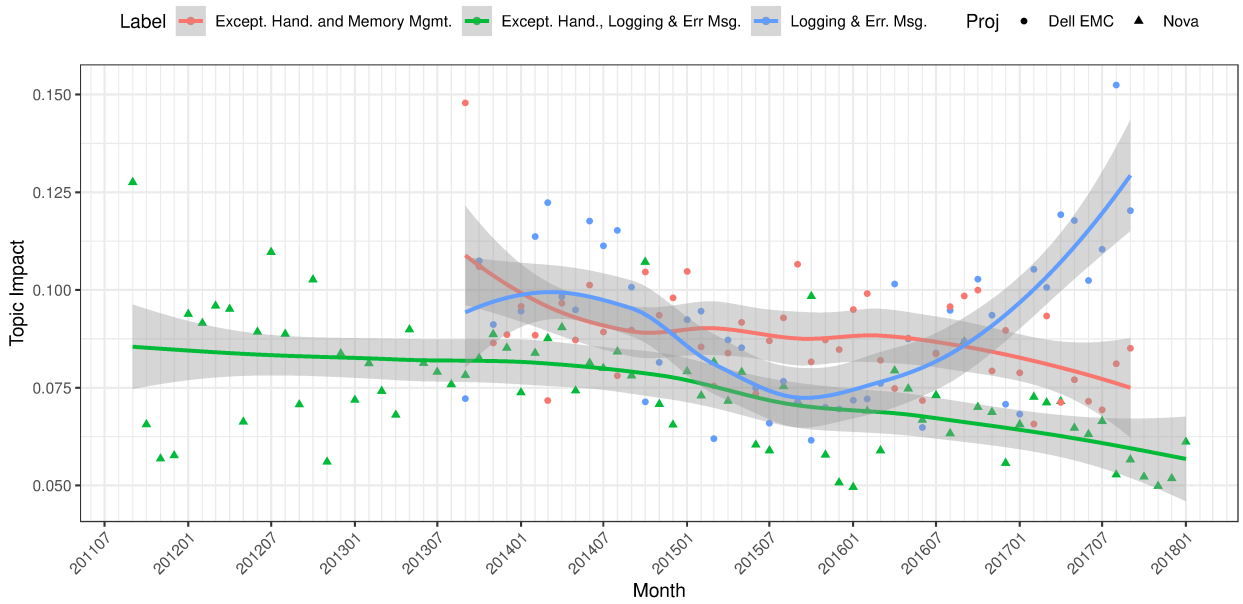


Figure 4.5: The impact score of topics discussing *Exception Handling* issues, plotted with regard to time.

for our LDA modeller to create an individual topic in this theme. On the other hand, Nova is mostly written in Python, a language with garbage collection built in. Thus, memory management is generally not an issue that developers need to discuss. Therefore, we only observe one topic in Nova related to *exception handling*.

Figure 4.5 shows that the *exception handling* topics (red and green lines) are decreasing over time. For both systems, it is crucial that system log messages are clear and concise. Indeed, system operators rely on the logs that are generated by these log messages to diagnose and recover from issues at runtime. Moreover, developers rely on these logs to debug software components. The decreasing trends may indicate that either the communities have become more adept at logging and exception handling over time, or that reviewers have put less effort into raising these concerns in recent time periods.

On the other hand, we observe a trough in the *logging and error messages* topic in Dell EMC before late 2015 and an increase afterwards. The trough and subsequent growth coincides with that the same influx of new developers and their onboarding process (cf. Observation 1), suggesting that the change of group dynamics could also affect the choice of discussion topics in the community.

Observation 5 — Although the Nova topic discussing language specific issues does not show a significant trend over time, the topic with this theme in Dell EMC shows a change of trend during 2015. Figure 4.6 shows

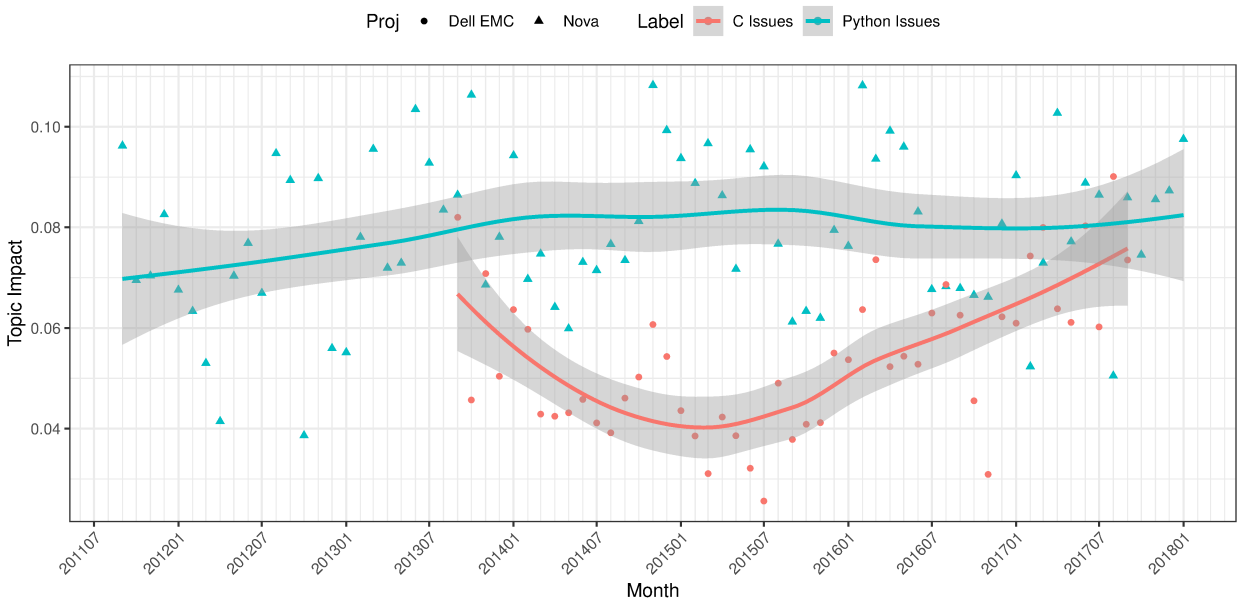


Figure 4.6: The impact score of topics discussing *Language Specific* issues, plotted with regard to time.

that one topic in the Dell EMC project and one in Nova that share the theme of *language specific* issues. The topic in Nova mainly involves detailed coding in Python, and does not show a significant trend over the observed period. However, in Dell EMC, the topic of string and buffer issues shows a change of trend during 2015. Similar to Observations 1 and 4, the change of trend closely coincides with the influx of new developers and their onboarding process, indicating again that changing group structures could affect the discussions in the community.

Observation 6 — Although the Nova topic of object-oriented design and the Dell EMC topic of function design show decreasing trends, the Dell EMC object-oriented design and concurrency topic shows an interesting change of trend in Q1 2016. Two topics in Dell EMC and one in Nova share the theme of *design*. Since design is crucial to the structure of a system, seeing that design-related topics are showing a decreasing trends in Figure 4.7 may raise concerns. For example, the *object-oriented design* topic in Nova and the *function design* topic in the Dell EMC project are showing consistent downward trends from the start to the end of the studied periods.

Figure 4.7 also shows that in early 2016, the *object-oriented design and concurrency* topic in the Dell EMC project shifts from an increasing trend to decreasing one. In addition, the decreasing *function design* trend flattens out. A development manager from the Dell EMC team explained that these trend changes

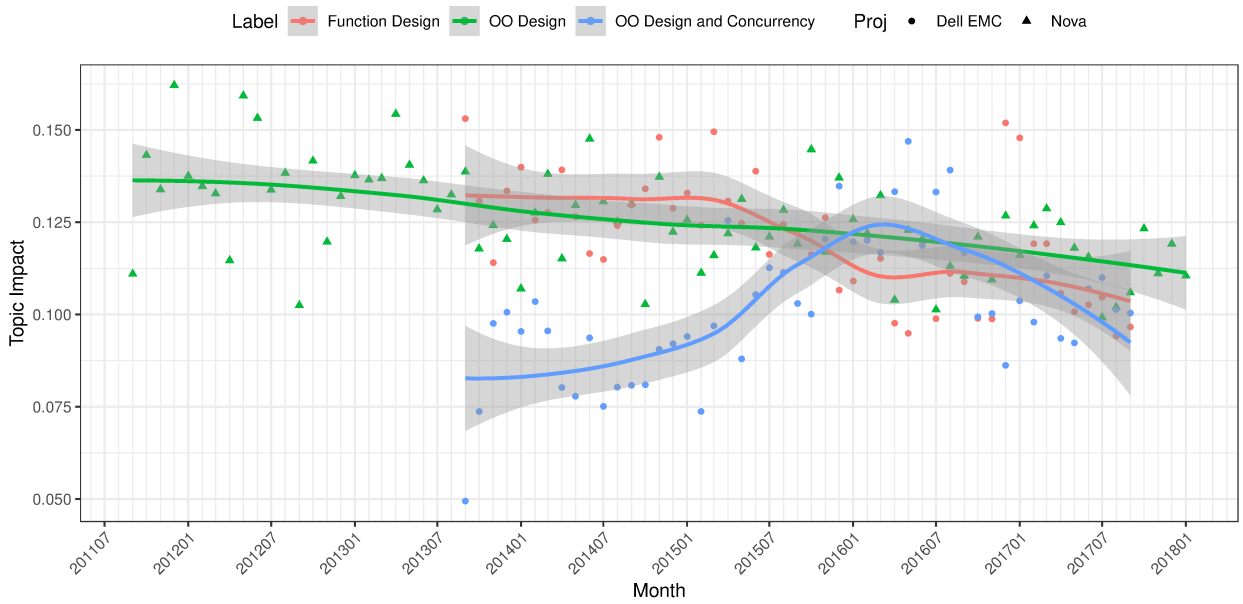


Figure 4.7: The impact score of topics discussing *Design* issues, plotted with regard to time.

coincide with a shift of a number of highly active staff members from an area of the codebase that is primarily implemented in object-oriented languages like C++ and Java to another area that is primarily implemented in procedural languages like C.

Observation 7 — Four out of seven context specific topics show increasing trends. Figures 4.8 and 4.9 show increasing trends for one of the three *context-specific* topics in the DELL EMC project (*Project Configuration*) and three of the four in NOVA (*API Issues*, *Provisioning Decision Making*, and *Volumes and Storage Management*). The significance of these trends is confirmed by Cox-Stuart trend tests. Indeed, in-depth, technical feedback (like that of the topics with increasing trends from above) is an expectation of participants in code reviews at Microsoft [6, 17]. One would hope to observe that the amount of feedback in these topics has been growing as a project has matured. Yet, three of the seven context-specific topics either show non-monotonic (*Project Terminology* in DELL EMC) or flat trends (*File Locations* in DELL EMC and *Virtual Machine* in NOVA) over time. Community focus can shift over time, so one should not expect that the impact scores for all context-specific topics should increase over time.

As discussed by Kononenko et al. [39], reviewers need to have a deep understanding of how the modified modules work in order to write comments that are associated with detail-oriented topics. Since writing comments in these topics may require more domain-specific knowledge from reviewers, it may

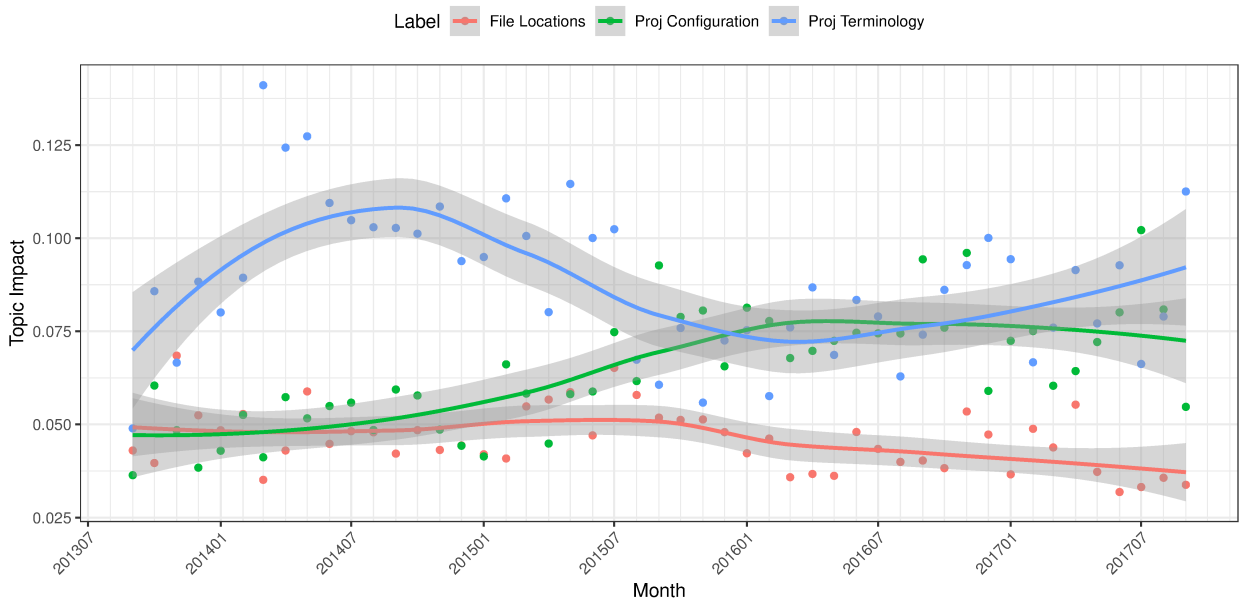


Figure 4.8: The impact score of topics in the Dell EMC project that are *Context Specific*, plotted with regard to time.

require more reviewer expertise. We discuss this observation further in RQ2.

Using our topic modelling approach, we can observe interesting trends in the reviewing behaviour of a community that often coincide with project events.

RQ2: How does the popularity of code review topics change as reviewers accrue experience?

RQ2: Approach

We analyze the relationship between reviewer experience and the prevalence of topics in their review feedback by (1) using two heuristics to estimate reviewer experience and (2) calculating the topic impact over different experience levels. Below, we define our two heuristics for estimating reviewer experience and the impact measure that we compute for varying levels of reviewer experience.

Reviewer Experience Heuristic by Core Level. In the OpenStack community, senior community members are recognized with core reviewer status. Core reviewers are the only team members who are permitted to provide reviews with a +2 or -2 score. In our first analysis, we evaluate whether there is a significant difference in the topics that core reviewers raise before and being promoted to core status. Since the Dell EMC project does not use a core reviewer mechanism, this analysis is only performed for

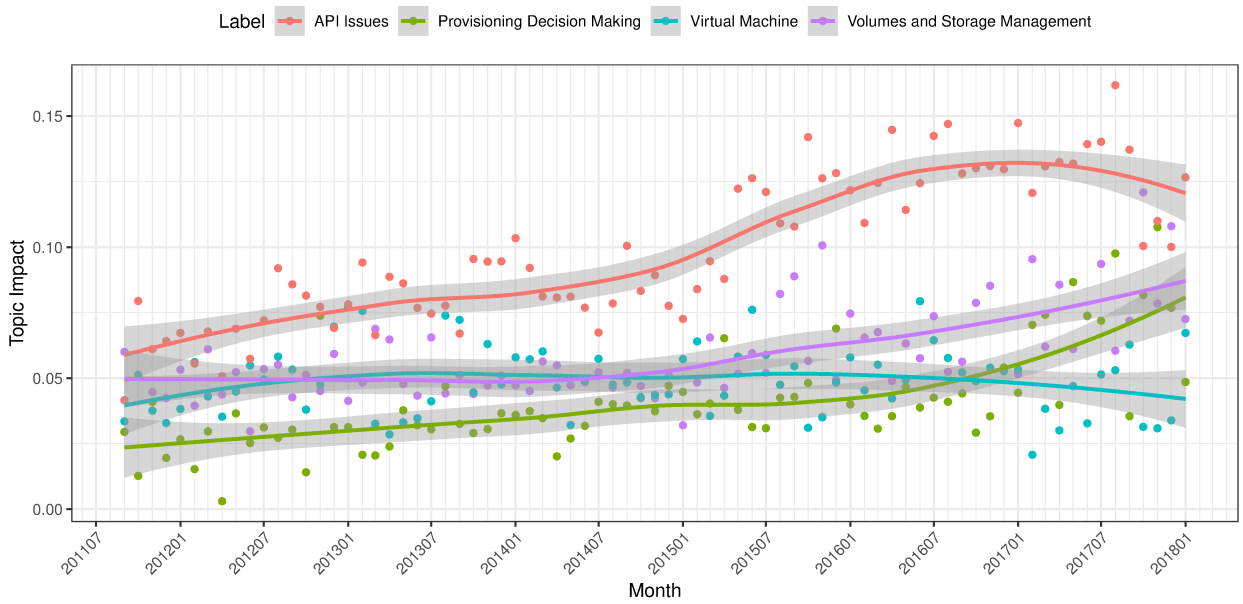


Figure 4.9: The impact score of topics in Nova that are *Context Specific*, plotted with regard to time.

Nova.

To perform our analysis, for each core reviewer in Nova, we need to split their reviewing data into before core promotion and after core promotion periods. Unfortunately, the Nova review database does not record the core promotion date. Hence, we estimate that date using the first +2 or -2 score that a core reviewer records. All reviewing activity before and after the estimated core promotion date is considered to be non-core and core activity, respectively.

Reviewer Experience Heuristic by Number of Reviews. We use the number of past review comments as a heuristic to estimate reviewer experience. For example, if a reviewer has written 100 comments prior to writing comment C, her experience score is 100 at the time that she writes comment C. This measure is based on an assumption that reviewers accumulate experience as they write more comments. Mockus and Herbsleb [51] found a link between developer expertise and the quantity of change that a developer has authored. Jiang et al. [36] have employed this concept in studying the relationship between developer experience and patch acceptance in Linux Kernel. Thongtanunam et al. [73] have shown that the concept extends to code reviewing activity. Thus, we believe that our assumption is sound. Indeed, past reviewing experience will provide a reviewer with a better overall understanding of the code review process and the projects that she is working on. Prior work suggests that more experienced reviewers tend to provide

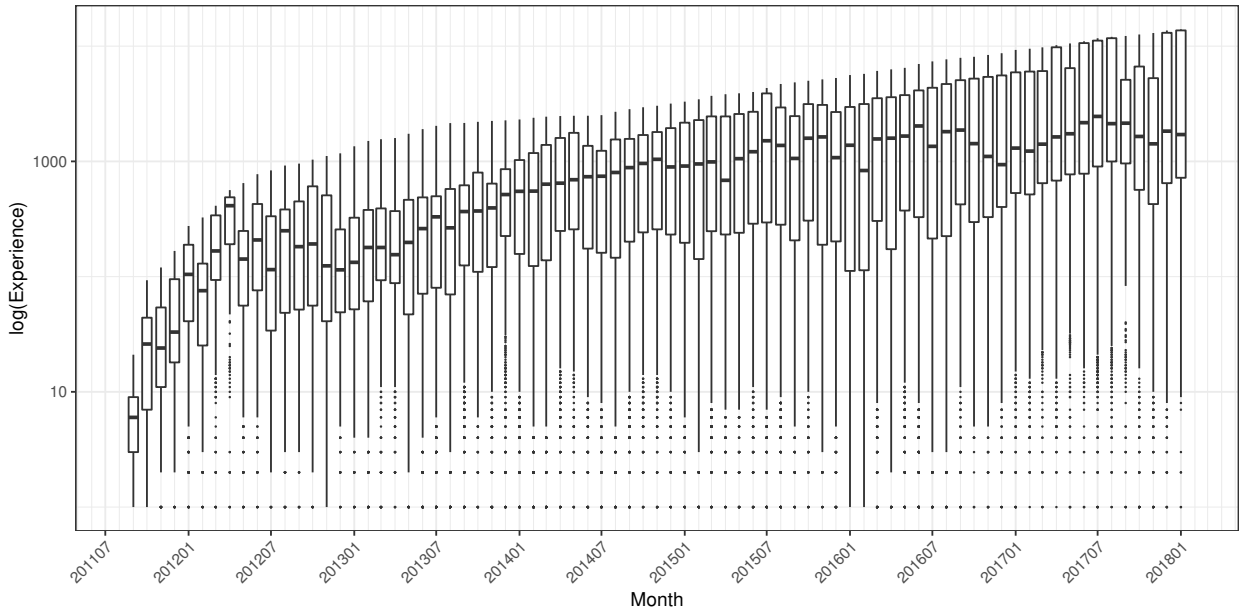


Figure 4.10: Experience scores (in log scale) of review comments in Nova for each month.

more useful feedback [17, 59].

In order to avoid oversampling the experience signal, we further group raw experience values into 100 levels of experience. Note that according to our defined heuristic, one reviewer's experience score (slightly) grows after each comment that they write. Figure 4.10 and Figure 4.11 show that in both projects, as time progresses, the breadth of experience scores grows, but the median and lower quartile values are relatively consistent. This results in heavily right skewed experience score distributions (i.e., there are far more inexperienced reviewers than experienced ones) in the studied systems. This is likely due to the often lamented high turnover rate in the software industry. An analysis has shown that 13.2% LinkedIn members in the software technology sector took new jobs at different companies in 2017.[‖] Another analysis shows that the employee retention rate ranged from 1.8 to 7.8 years in 15 large Bay Area technology employers.[**] Because of the skewness, grouping experience values using equidistant thresholds will undersample the low experience values and oversample the high experience values. Therefore, we group them into bins that contain an equal number of reviewers. We select 100 as the number of bins by analyzing the distribution of the data. Analysis of the data set with 50 and 75 bins suggests that the general direction and shape

[‖] https://bit.ly/2pfQdsc
[**] https://bit.ly/2MPqGRr

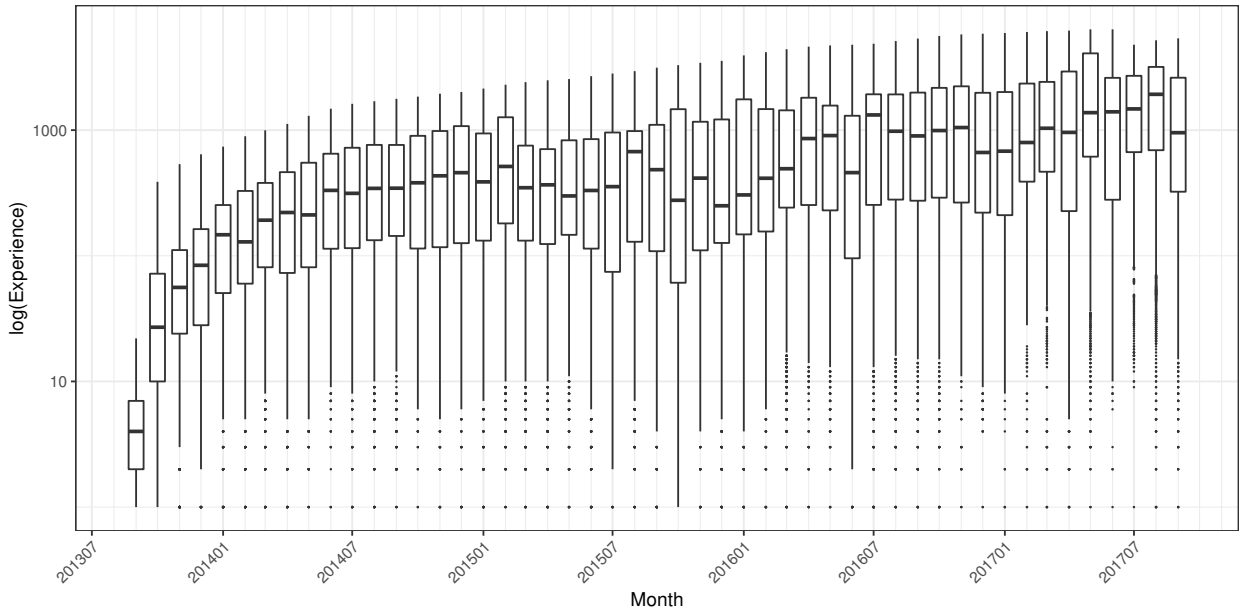


Figure 4.11: Experience scores (in log scale) of review comments in Dell EMC for each month.

of the distribution remain consistent. We include the figures of the 50- and 75-bin experiments in our replication package. Nevertheless, we elaborate on this threat to validity in Section 4.6.

Topic Impact by Experience Levels or Core Levels. Similar to the analysis of how topic popularity evolves over time in RQ1, we compute *topic_exp_impact*—a measure that captures the prevalence of a topic for reviewers with a given level of experience, or whether or not the reviewer has been promoted to the core team yet. We redefine the topic impact measure (Equation 4.2) to focus on experience and core membership levels rather than time periods. More specifically, we define the experience impact of a topic t_k in an experience or core membership level x as:

$$topic\_exp\_impact(t_k, x) = \frac{1}{|D(x)|} \sum_{\substack{\forall d_i, d_i \in D(x), \\ (d_i, t_k) \geq 0.1}} \delta(d_i, t_k) \tag{4.3}$$

where $D(x)$ represents the set of review comments that are written by reviewers with experience or core membership level x. The *topic_exp_impact* measures the proportion of review comments that have a non-negligible association with topic t_k (i.e., $\delta(d_i, t_k) \geq 0.1$) for reviewers with experience or core membership level x.[††]

[††]Similar as in RQ1, a sensitivity analysis that explores threshold values between 0.1 and 0.2 yielded no significant change in

Table 4.3: A table with the p-values of the Mann-Whitney U test and the effect size for the comparison of core/non-core reviewers in Nova.

Topic Label	M.-W. p-value	Effect Size
Volumes and Storage Management	0.73	negligible
Exception Handling, Logging and User-facing Error Msg	0.86	negligible
Provisioning Decision Making	0.22	small
Code Review Process and Minor Issues	0.38	negligible
Virtual Machine	0.84	negligible
Object Oriented Design	0.48	negligible
Unit Testing	0.53	negligible
Python Collections	0.26	small
API Issues	0.04	small
Code Style	0.2	small

For topic impact over core membership levels, Figure 4.12 compares core and non-core topic impact scores for each topic. Each of the data point in the box plot represents the topic impact of a reviewer during her time before and after being promoted to the core team. We test if there is a significant difference between these topic impact scores using paired, two-tailed Mann-Whitney U tests ($\alpha = 0.05$). Since we are conducting multiple comparisons between samples drawn from the same original population, we apply Bonferroni correction, which adjusts the $\alpha = \frac{0.05}{10} = 0.005$. To estimate the effect size of the difference, we use Cliff's delta, which is negligible when $0 \leq \text{delta} < 0.147$, small when $0.147 \leq \text{delta} \leq 0.33$, medium when $0.33 \leq \text{delta} \leq 0.474$, and large otherwise.

For topic impact over experience levels, we show key plots of the trends over experience levels. Similar to RQ1, the plots contain raw data points and Loess-smoothed regression lines. We also apply the two-tailed variant of the Cox-Stuart trend test to all of the topic experience trends ($\alpha = 0.05$).

RQ2: Results

Observation 8 — We do not observe any significant difference in the behaviour of reviewers before and after promotion to the core team. As shown in Table 4.3, the Mann-Whitney U test results do not support rejection of the null hypothesis that both samples follow the same distribution. This may be a result from the small number of core reviewers, as only 41 of the 992 Nova reviewers are members of the core team.

the shape of the topic trends. See the online replication package for more detail.

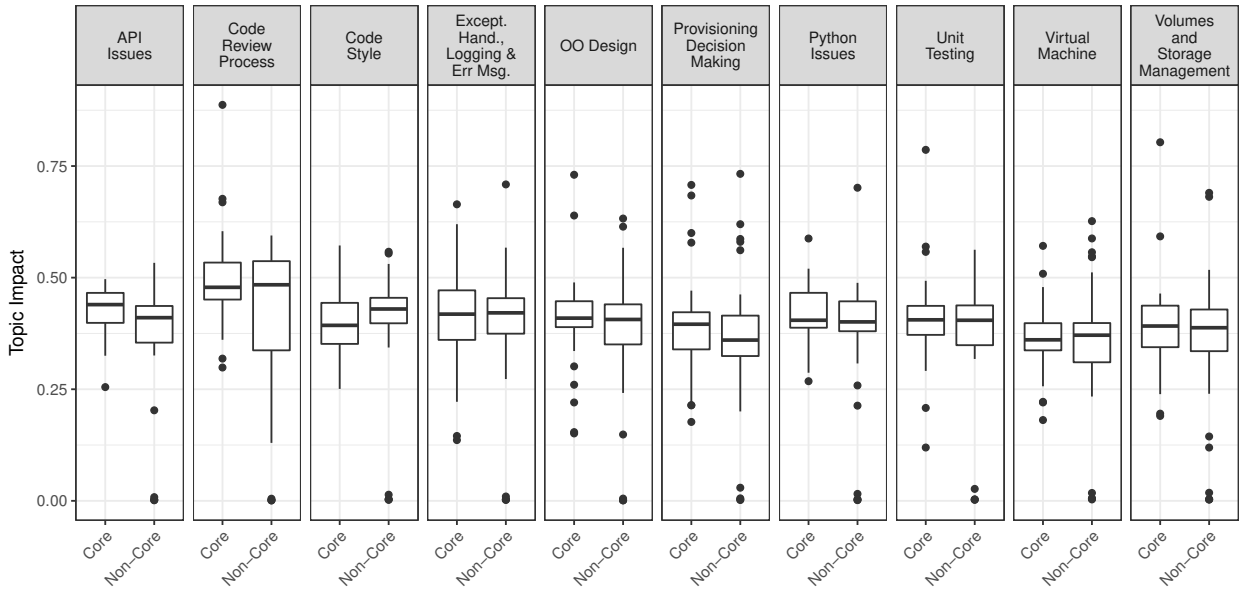


Figure 4.12: The comparison between core and non-core reviewers' topic impact scores in Nova.

Indeed, we have only 41 non-core and core observation pairs, which may not yield the statistical power necessary to reject the null hypothesis.

Observation 9 — As reviewers accrue experience, trends in code style topics diverge among the studied projects Figure 4.14 shows that after a common flat period, the experience trend for *code style* tends to decrease in Nova and increase in the Dell EMC project. On the surface, code style feedback seems to have a low ROI, and having experienced reviewers spend their effort on it seems wasteful. Thus, the decreasing trend for Nova is encouraging, and the increasing trend for the Dell EMC project may raise concern. When we discussed our observations with the Dell EMC staff, a manager from the Dell EMC project explained that the team had discussed a growing need to address code style problems in team meetings. Hence, the observation that the rate of code style concerns being raised increases with experience seemed to match his expectation.

Observation 10 — The code review process topic tends to decrease as reviewers accrue experience. Figure 4.13 shows a downward trend of topic impact in *code review process* discussions for both Dell EMC and Nova reviewers when they become more experienced. Developers at Microsoft [17] and Mozilla [39] argue that, when trying to improve their changes, code review process comments are not as helpful as the more context-specific comments. The observed trends for the *code review process* topics of both studied

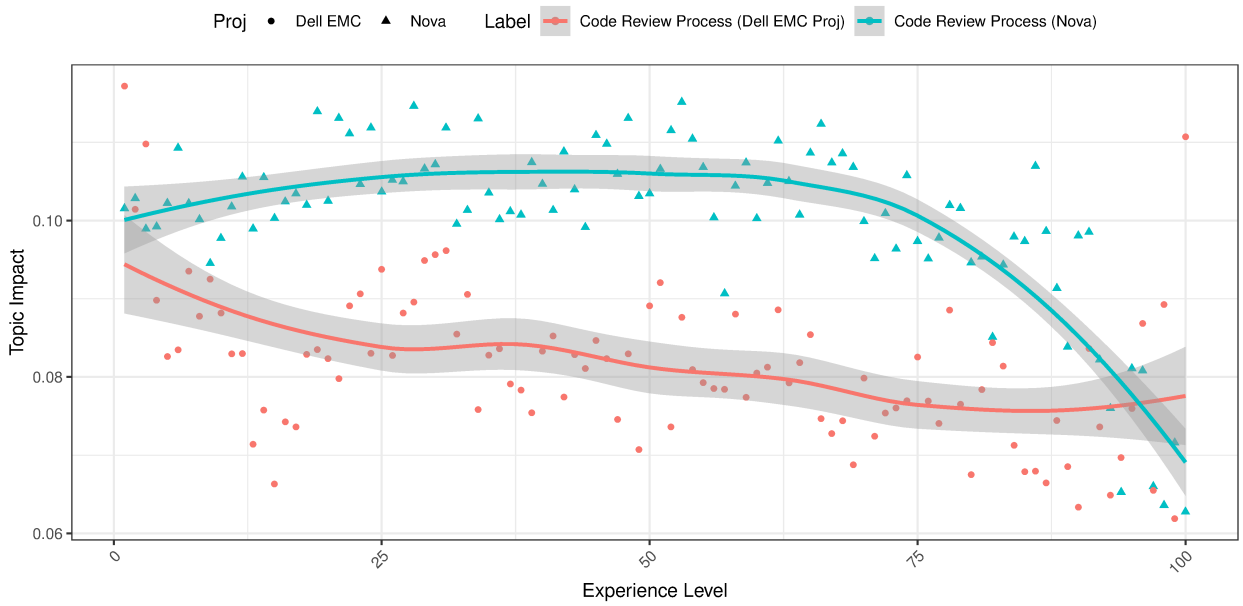


Figure 4.13: The impact score of topics discussing *Code Review Process* issues, plotted with regard to reviewing experience.

projects suggest that most of the experienced reviewers provide fewer process-related details in their review feedback.

Observation 11 — Although two of the three exception handling topics show a flat trend, the logging and user-facing error messages topic the Dell EMC project shows an increasing trend as reviewers accrue experience. Figure 4.15 shows that while the *exception handling* topics in Nova and the Dell EMC project have relatively flat trends, the Dell EMC topic on *logging and user-facing error messages* shows a rapidly increasing trend towards the upper end of the experience scale. Similar to the RQ1 observations about *exception handling* topics, the increasing trend in logging and user-facing error messages after late-2015 coincides with the dedication of additional effort to generating more meaningful event logs and error messages to improve the operators' experience when maintaining and debugging issues with the Dell EMC project.

Observation 12 — Although there is no significant trend for the object-oriented design topic in the Dell EMC project, we observe a decreasing trend for OO design and concurrency in Nova and function design in Dell EMC with regard to reviewers' experience. Figure 4.16 shows that the *object-oriented design* topic in Nova is less frequently discussed, while that in Dell EMC does not show a significant trend when reviewers accrue more experience. It may be possible that because the basics of object-oriented

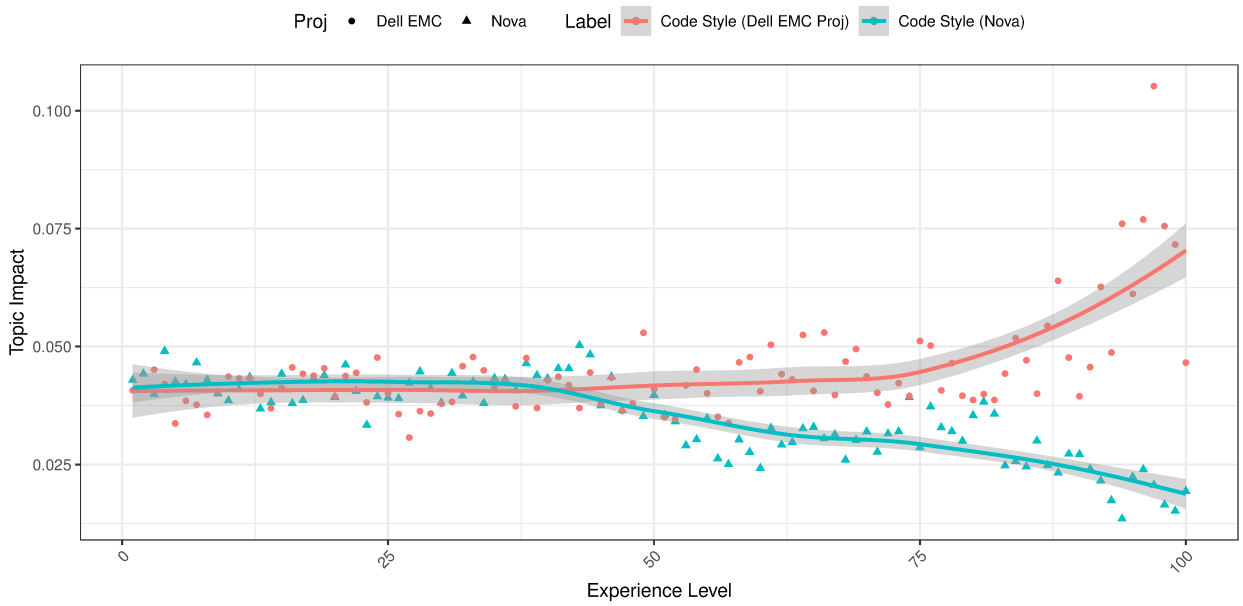


Figure 4.14: The impact score of topics discussing *Code Style* issues, plotted with regard to reviewing experience.

programming is a must-know for most developers in Nova, the entry requirement is relatively lower than the other (project-specific) topics. Meanwhile, because *function design* topic contain lower-level issues, reviewers with less experience in the project are able to provide constructive feedback. Community gurus may not be needed to provide low-level *function design* feedback. Thus, reviewers with less project experience tend to raise context-agnostic issues on topics like *object-oriented design* and *function design*.

Observation 13 — Reviewers with more project experience tend to provide more language-specific and context-specific feedback. Figure 4.17 shows an increasing trend for both language-specific topics in Nova and Dell EMC. It seems natural that developers would accrue knowledge in relevant language features as they gain experience working on the studied projects. Moreover, Figure 4.18 shows that the more experienced Dell EMC reviewers provide *project configuration* feedback (a *context-specific* topic) more frequently, contributing to the community by using their extensive knowledge in the Dell EMC project. Similarly, we also observe a surge in two of the four *context-specific* topics in the Nova community in Figure 4.19.

Using our topic modelling approach, we can observe that experienced reviewers may have different reviewing focus according to the need of different communities.

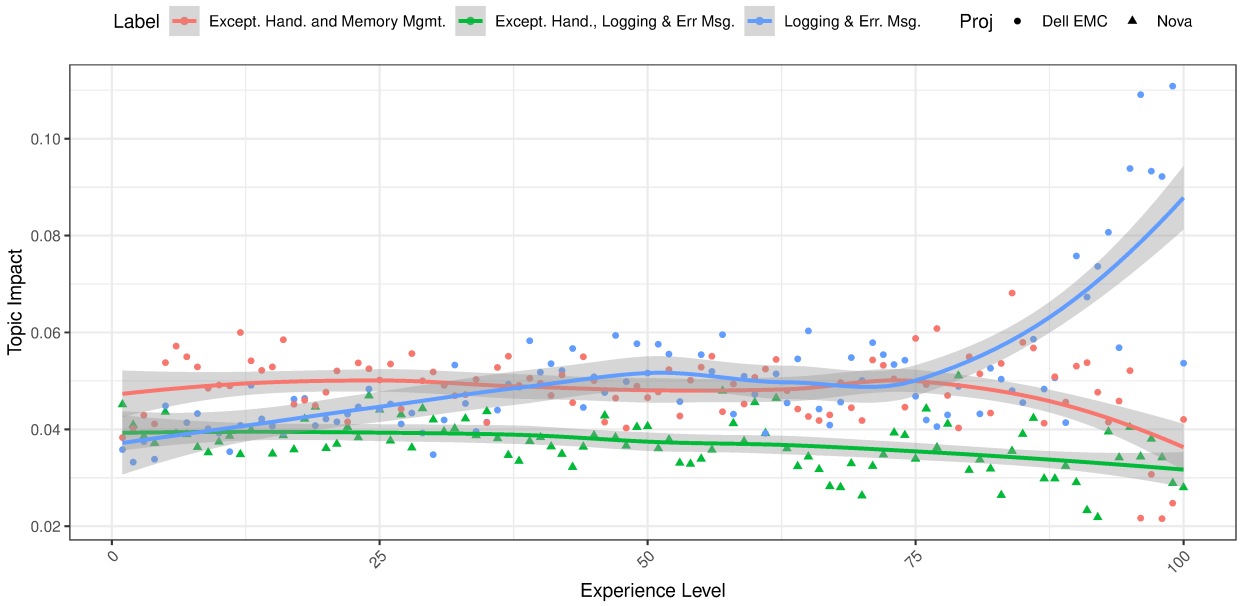


Figure 4.15: The impact score of topics discussing *Exception Handling* issues, plotted with regard to reviewing experience.

4.5 Practical Implications

We now discuss the broader implications of our observations.

4.5.1 Monitoring Community and Reviewer Skill Development

We believe that our topic models lay the necessary groundwork for a code review analytics framework. Our results from RQ1 show that the prevalent topics in code reviewing discussions change as both communities have become more mature. An analytics tool that analyzes quantitative data from topic models like ours could be used to track changes in community and reviewer focus. These changes could be compared with qualitative data that is gathered from the development team to detect whether the reviewing process is evolving to fulfill community needs. In addition, reviewers could provide personal goals and check their progress over time with respect to the quantitative data from the topic models.

4.5.2 Mentorship Programs in Code Review

Our findings suggest that there exists a gap between novice and experienced code reviewers. Results from RQ2 show that, while the specific topics that yield significant trends with respect to experience vary from

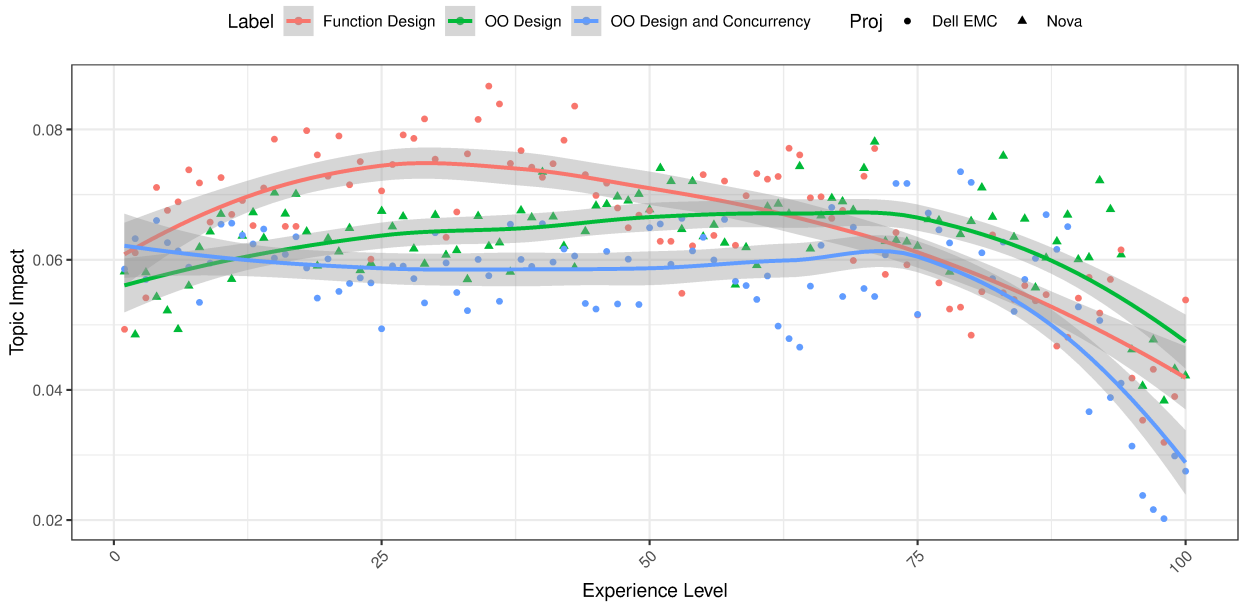


Figure 4.16: The impact score of topics discussing *Design* issues, plotted with regard to reviewing experience.

project to project, the prevalent topics in code reviewing discussions differ across experience levels.

Knowledge transfer is one of the main benefits of code review [6]. An explicit mentorship program may help novice reviewers to develop their reviewing skills more quickly. With additional knowledge about the feedback that experienced (top) reviewers provide, the novice may accumulate experience and produce more useful review comments more quickly, without the need to accrue months or years of reviewing experience.

4.6 Threats to Validity

In this section, we discuss the threats to the validity of our study.

4.6.1 External Validity

Threats to external validity have to do with the generalizability of our results to other subject systems. We study projects from the OpenStack and Dell EMC communities, which have embraced and invested in their code reviewing process (see Section 4.2.1). Our results may not generalize to code review processes of other software communities. Since OpenStack is open source, the community is comprised of

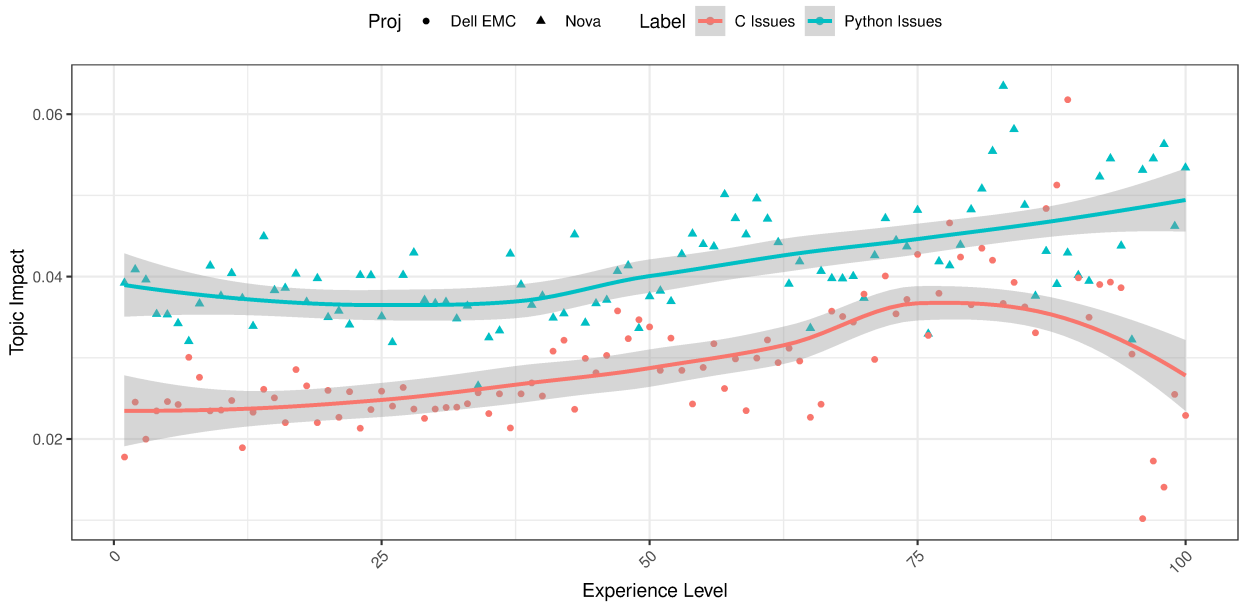


Figure 4.17: The impact score of topics discussing *Language Specific* issues, plotted with regard to reviewing experience.

a broad spectrum of developers, including hobbyists and professional developers. Moreover, the professional developers represent several companies, each with its own corporate culture (e.g., IBM, NEC). In addition to OpenStack, the Dell EMC community provides valuable insight into code review practice in a more traditional, single organization industrial setting. Although we are only analyzing two software communities, the sample of developers and development cultures is rich and diverse. Nonetheless, additional replication studies will likely yield further insight.

4.6.2 Internal Validity

Threats to internal validity have to do with whether other plausible hypotheses could explain our results. Our work focuses on comments that are recorded in code review platforms as an indicator of the discussions that take place during code review. Although the OpenStack and Dell EMC communities tightly integrate their code review platforms into their development cycle, in-person meetings or other communication media (e.g., e-mail) could be used to supplement the discussions that are happening on the platform. Unfortunately, explicit links between code changes and other communication channels are scarce, and recovering these links is a non-trivial research problem [7, 14].

We filter out comments that are not natural langauge (code) before building topic models using NLoN,

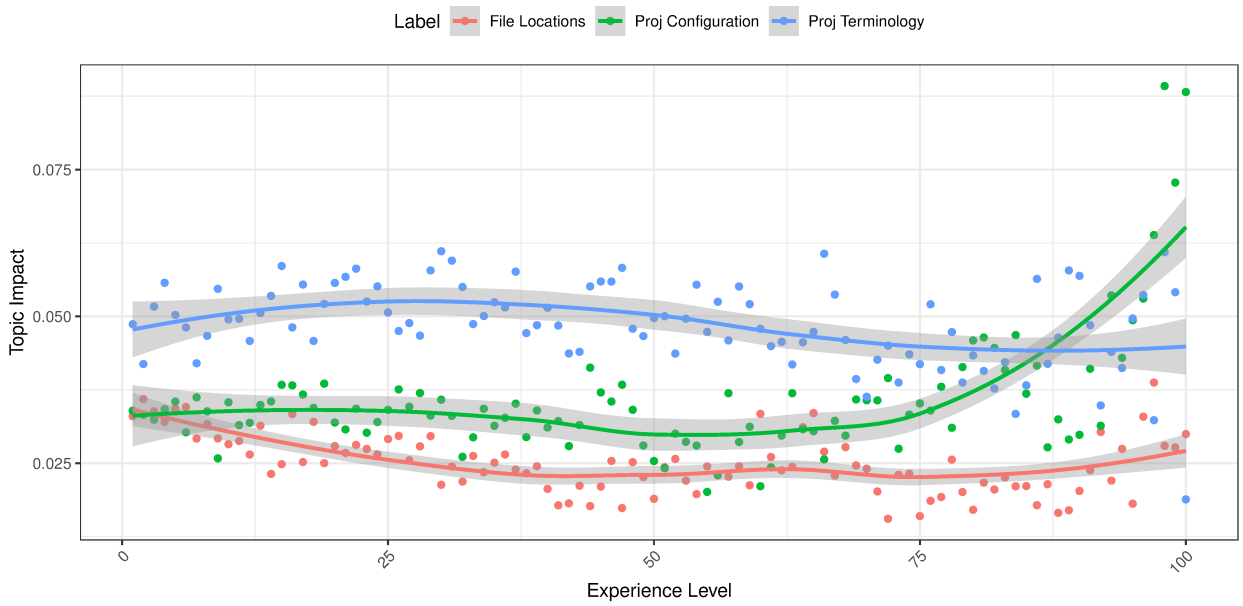


Figure 4.18: The impact score of topics in the Dell EMC project that are *Context Specific*, plotted with regard to reviewing experience.

a state-of-the-art machine learning package [44]. However, NLoN is not able to identify comments that are a mixture of natural language and code. Because of this, some of the comments in the dataset may contain some code elements. Although some of the code tokens are also natural language elements (e.g. `device_owner`, `project_status`), the results may be more accurate with all code snippets removed. A more powerful means of filtering code snippets is a complex research problem in and of itself, and we plan to explore it further in future work.

We assume that the changes in topic impact are due to changes in time period or reviewer experience; however, confounding factors may play a role. For example, the size of the backlog of tasks that a developer is working on may also impact the type of feedback that reviewers can provide. We plan to explore this and other potential confounding factors in future work.

4.6.3 Construct Validity

Threats to construct validity have to do with the alignment of our choice of indicators with what we set out to measure. Borrowing from the underlying concept of the user experience metric proposed by Jiang et al. [36], our experience heuristics are based on the number of prior review comments in either dataset, or whether a reviewer has given +2/-2 scores on patches in Nova. Core reviewers may have not given +2/-

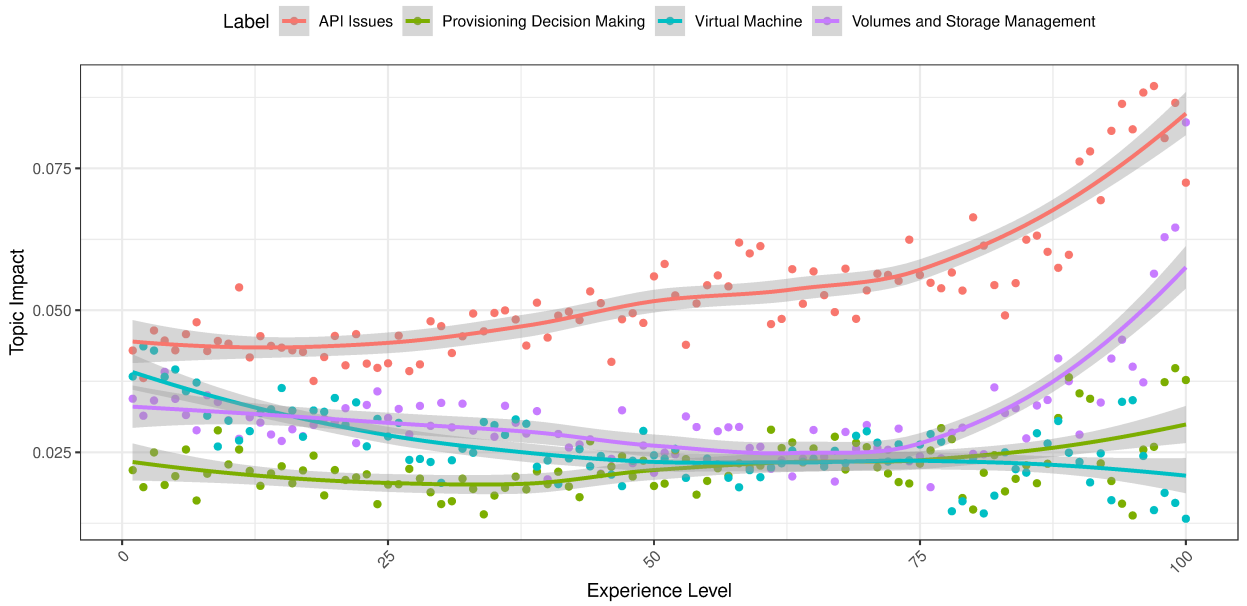


Figure 4.19: The impact score of topics in Nova that are *Context Specific*, plotted with regard to reviewing experience.

2 scores in the dataset, rendering false negatives when we detect core reviewers in Nova. Moreover, since the code review data for the Dell EMC project is an excerpt from their two-decade-long development history, some of the reviewers' past experience in the project is omitted. Reviewers may gain experience from other channels, e.g., development and review activities in other projects. Since we are not aware of a metric that can be computed to accurately reflect overall developer experience, we use an experience heuristic that we can measure using the data that we have on hand. If a more accurate experience heuristic is discovered, future replication studies could prove fruitful.

Our naming of topics is based on the authors' subjective understanding of the context. Other readers may arrive at different names of topics from the presented word list and review comment samples. Some of our analysis could be skewed given another form of abstraction for the topics. For naming the topics, both authors individually read the list of top word related to the topic, as well as the related review comments in order to reach a consensus about on the topic name. Moreover, the topics and concepts that we detect share similarities with those found in other contexts that were reported in the literature [6].

To conduct our experiment, we need to select settings for: (1) the number of topics in our LDA model ($K = 10$), (2) the time units for RQ1 (months), and (3) the number of experience levels for RQ2 (100). We have experimented with tuning parameters, and maximized the stability of our topic model, making

results more replicable. However, selecting different settings may still yield different results. The goal of this study is not to identify the optimal settings for these experimental parameters, but rather to examine whether and how reviewing feedback changes as communities and reviewers mature.

4.7 Chapter Summary

Code review is a lynchpin of most modern software quality approaches. To derive value from a code review process, it must produce valuable feedback for authors to address. While past work [45, 11] has explored the issues that are found and fixed during the code review process, little is known about how reviewing feedback changes as a software community and its stakeholders mature.

In this chapter, we train topic models to identify latent topics in the review comments of an open source and a proprietary community: OpenStack Nova and a Dell EMC project. Through a longitudinal study of 248,695 review comments in 39,249 changes in two projects, we make the following observations:

- The change of reviewing behaviour of a code review community often coincides with project events.
- Experienced reviewers who grow up in different code reviewing communities focus on different topics according to project needs.

4.7.1 Concluding Remarks

The focus of this chapter is on studying the evolution of code reviewing feedback topics, providing insights of the change of focus for the teams as a whole. On the other hand, we would like to help individual developers make better decisions in investing code reviewing effort. To that end, in the next chapter, we introduce *BLIMP Tracer*, a build impact analysis system that we integrated with the code reviewing platform at a Dell EMC project team.

An earlier version of the work in this chapter appears in Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME 2018) [80].

5

Code Review with Build Impact Analysis

5.1 Introduction

Code review refers to the practice where fellow developers inspect code changes and provide feedback to the author. Dedicated code review tools that manage the modern code review process have become a commonplace in practice. These tools allow developers to post patches and select relevant reviewers to inspect their patches. The review process itself is a valuable practice that development teams use to ensure software quality [48], improve team communication [6], and leverage team problem-solving capacity [61].

However, the mere existence of a code review does not improve code quality. To truly improve the quality of a patch, reviewers must consider the potential implications of the patch and engage in a discussion with the author. Prior work shows that a lack of reviewer participation is correlated with a drop in software release quality [49, 70] and a drop in design quality [52].

On the other hand, rigorous code review introduces overhead on developers, whose time is a limited and valuable commodity. Bosu and Carver [16] find that developers spend an average of six hours per week reviewing code. The time spent reviewing code is an expensive context switch from other important development tasks (e.g. repairing and improving code). Making matters worse, patch authors at Microsoft report that an average of 35% of code review comments are not useful [17], suggesting that a large proportion of reviewing time may be misspent generating feedback that is not valuable.

Since some changes are of greater risk than others, some patches will require a more rigorous review than others. Czerwonka et al. [25] argue that spending an equal amount of reviewing effort on all code patches is a suboptimal use of development resources. Currently, to reduce waste in the reviewing process, developers use their intuition and their past experience to decide which patches require detailed feedback. However, knowing which patches require more reviewing attention than others is a difficult problem for code authors and reviewers alike.

An understanding of the impact that a patch has on the entire software system may help stakeholders to focus on reviewing effort on patches that have a broader impact on the project. More specifically, we believe that: Patches that impact mission-critical project deliverables or deliverables that cover a broad set of products should involve more reviewing investment than others. However, such information is missing from modern code reviewing interfaces.

To understand the impact that a patch will have on a system, Change Impact Analysis (CIA) techniques have been proposed [5]. However, recent work suggests that CIA techniques are rarely adopted in practice [42]. To understand the state of CIA within the studied product team at Dell EMC, we conducted a preliminary survey of 45 developers. In the survey, we ask developers how they assess the potential impact of a patch. The results indicate that, despite their tendency to produce fault prone and incomplete results, developers choose to use command line tools, such as `grep` and `find` to estimate the impact of changes (likely due to their ubiquity and flexibility), to complement their intuition-based on prior knowledge of the modified files. Indeed, as Li et al. [42] reported, dedicated and commercialized use of CIA tools is rare. Use of ad hoc and intuition based approaches may lead to false positives (i.e. patches that did not need to be reviewed rigorously, but were) or false negatives (i.e. patches that should have been reviewed

rigorously, but were not).

To help reviewers make deliverable-based decisions of reviewing where to invest reviewing effort, we developed Build Impact (BLIMP) Tracer, an impact analysis system that we integrated with the code review platform of the studied product team at Dell EMC. Unlike traditional change impact analysis [5], BLIMP Tracer exposes the impact that a change has on project deliverables rather than other areas of the source code. This difference is of key importance in the context of the studied Dell EMC team because the subject system is comprised of several deliverables that belong to several customer-facing products. In a nutshell, BLIMP Tracer produces impact analysis reports by first extracting the Build Dependency Graph (BDG) of the system, then recursively traverses the graph starting from the changed files to identify the (set of) impacted product deliverables.

To evaluate BLIMP Tracer, we deployed it within the production reviewing environment at the studied Dell EMC team. We conducted a comparative user study with five developers. More specifically, we solicit feedback from participants during their use of BLIMP Tracer, and compare it with current style of conducting impact analysis. In all five cases, BLIMP Tracer improves the speed and accuracy of locating impacted software components of a code patch. In addition, we find that BLIMP Tracer provides developers with a clearer understanding of the project build-time architecture [78], which will likely help when onboarding newcomers to the project.

5.1.1 Chapter Organization

The remainder of this chapter is organized as follows. Section 5.2 describes the code review process at the studied Dell EMC team and shows a motivational example. Section 5.3 describes the design and results of the preliminary survey distributed to the developers. Section 5.4 describes the design of BLIMP Tracer. Section 5.5 discusses the design of our user study. Section 5.6 presents the results with respect to our user study. Section 5.7 describes the threats to the validity of our study, and finally, Section 5.8 draws conclusions.

5.2 Background

In this section, we present a motivating example to demonstrate the value of BLIMP Tracer.

5.2.1 A Motivational Example

Adam, a new developer who has just started working at Dell EMC one week ago, is submitting his first patch on Review Board. As is part of the Dell EMC code reviewing practice, Adam has to indicate which team members to invite to review his patch. Being a new team member, Adam does not yet know which members of the team have the necessary expertise in the areas of the code base that he modified. Thus, he relies on the Review Board recommendations to select his team lead, Becky, as a reviewer.

Being a team lead, Becky receives plenty of review requests. In order to have time to complete her other tasks, she needs to be selective where she focuses her reviewing effort. She needs to decide whether to apply her full effort to the review (high time cost) or perform a quick review (low time cost).

At the time when Becky is notified of Adam's review request, she already has a backlog of ten review requests. All ten of the review requests are associated with issue reports of equal severity and priority. Based on her intuition, Becky decides to prioritize the patches that are larger in size because she believes that larger patches are inherently more risky. However, Becky's decision may not be optimal because Adam's patch involves a change to broadly adopted in-house libraries. Changes to those libraries may be inherently more risky than large changes because they impact a large amount of customer-facing functionality, ending up being linked with several project deliverables.

BLIMP Tracer is designed to provide decision support for Becky. She can consult the results of BLIMP Tracer for Adam's patch, and determine which deliverables may be impacted by his patch. By exposing the impacted deliverables of a patch, Becky can reason about the impact of a change on customer-facing products and functionality. Knowing which products are impacted by a patch helps Becky make a more informed decision about how to prioritize her backlog of patches for reviewing.

5.3 Preliminary Survey

In this section, we present the design of a developer survey that we conducted at Dell EMC. Through the survey, we aim to gain a better understanding of how developers conduct impact analysis. The questions in the survey were intentionally open ended to allow for developers to explain their current practices in language that is natural to them.

5.3.1 Survey Design

At a high level, the survey is composed of three parts. The first part focuses on information about the developer, with demographic questions about their general software development experience and their experience working on the product suite at Dell EMC. The second part asks developers about their prior knowledge in software change impact analysis. The third part asks developers to reflect on their day-to-day procedures for assessing the impact of patches. We also invite developers who are interested in further discussing their experiences to participate in follow-up interviews afterwards.

The survey was broadcasted to 45 on-site developers, 12 of whom responded (27% response rate). The survey was also broadcasted to off-site developers, four of whom responded.

5.3.2 Demographic Information

The 16 respondents of this survey work in development offices in Canada and India. Figure 5.1 shows that the respondents have a broad range of experience with the Dell EMC product suite, ranging from one to 18 years, with a median of 6.5 years. Similarly, the respondents' experience on the current product varies from zero to 18 years, with a median of 6.5 years. On the other hand, the majority of the respondents are senior software engineers, with general software development experience ranging from one to 29 years, with a median of 14 years. Indeed, twelve of the respondents have more than five years of experience. The wide span of experience of the responding developers within Dell EMC assures that the observed results apply to a broad range of development backgrounds. The large number of senior developers ensures that our results are not biased towards new developers who may not have formed impact analysis habits.

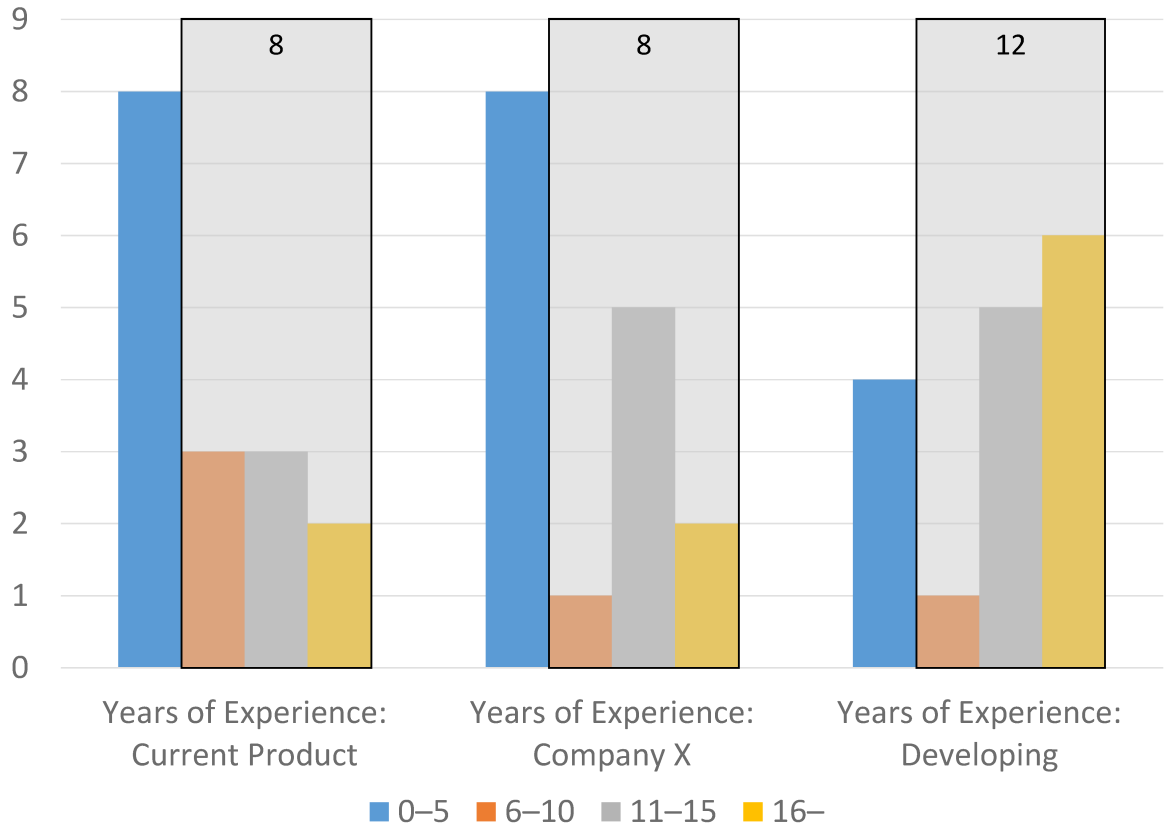


Figure 5.1: The survey respondents' experience in software development in multiple contexts. The veterans (with more than five years of experience) are shadowed in gray.

5.3.3 Change Impact Awareness

Figure 5.2 shows that although most (nine of the 16) respondents claim they would first apply the code patch under review and execute a project build job, build impact assessment is also the first thing code reviewers do when they start reviewing a code patch. The process of applying code patches under review to software systems and building them can be automated. In fact, the studied product team at Dell EMC employs an automated tool that applies code patches, builds the project, and reports whether the build breaks after applying the patch (recall Step 2 from Section 2.1). This tool is integrated with the code review platform and posts a comment indicating whether the patch introduces build problems once the result is generated. However, this tool does not provide feedback on the build impact of the code change, the other important step that developers take into consideration before performing a code review.

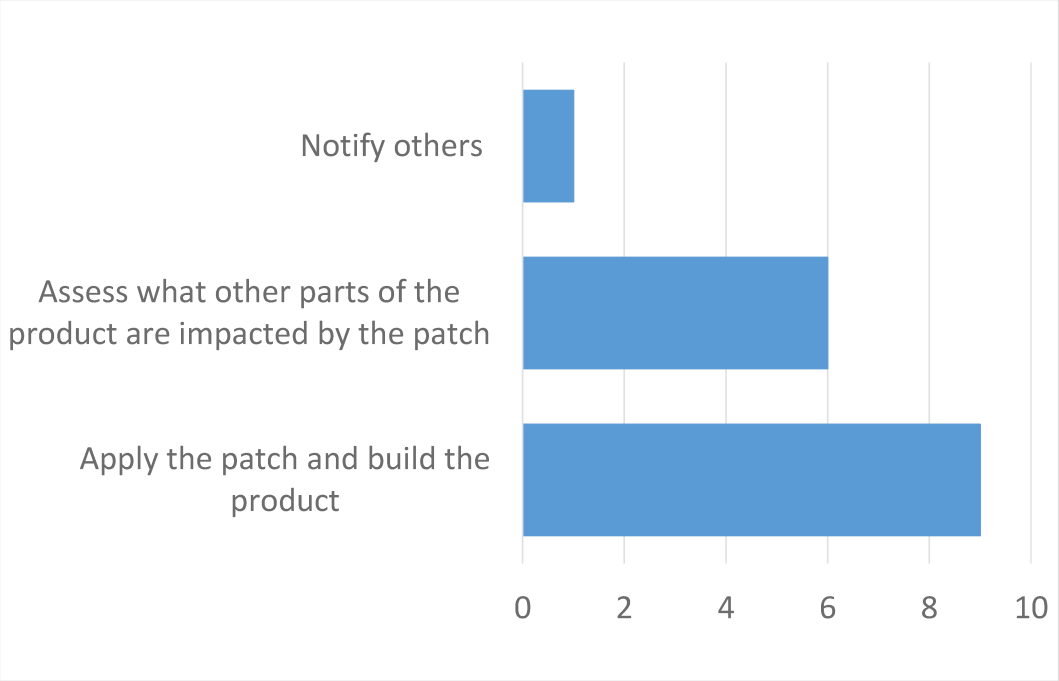


Figure 5.2: The number of developers on what they would first do when they start reviewing patches.

Meanwhile, we find that six of the 16 respondents acknowledge that they would check what part of the software project the code patch impacts as the first step of code review. Indeed, one developer stated that understanding the impact of a code patch is so important that 70–80% of his reviewing time is spent performing impact analysis. He explained that *"The small chunk of code that an engineer author in a change should already be examined before submitting (to Review Board), so there should not be a problem there. The real problems come with the files that depend on the change."*

Although a number of community members recognize the importance of impact analysis, they do not use any specialized tools for it and often resort back to using command line tools to investigate the impact of a patch. In fact, only two out of 16 survey respondents declare that they have used impact analysis tools. Without dedicated impact analysis tools, using other methods for finding the impact of code patches are relatively time-consuming.

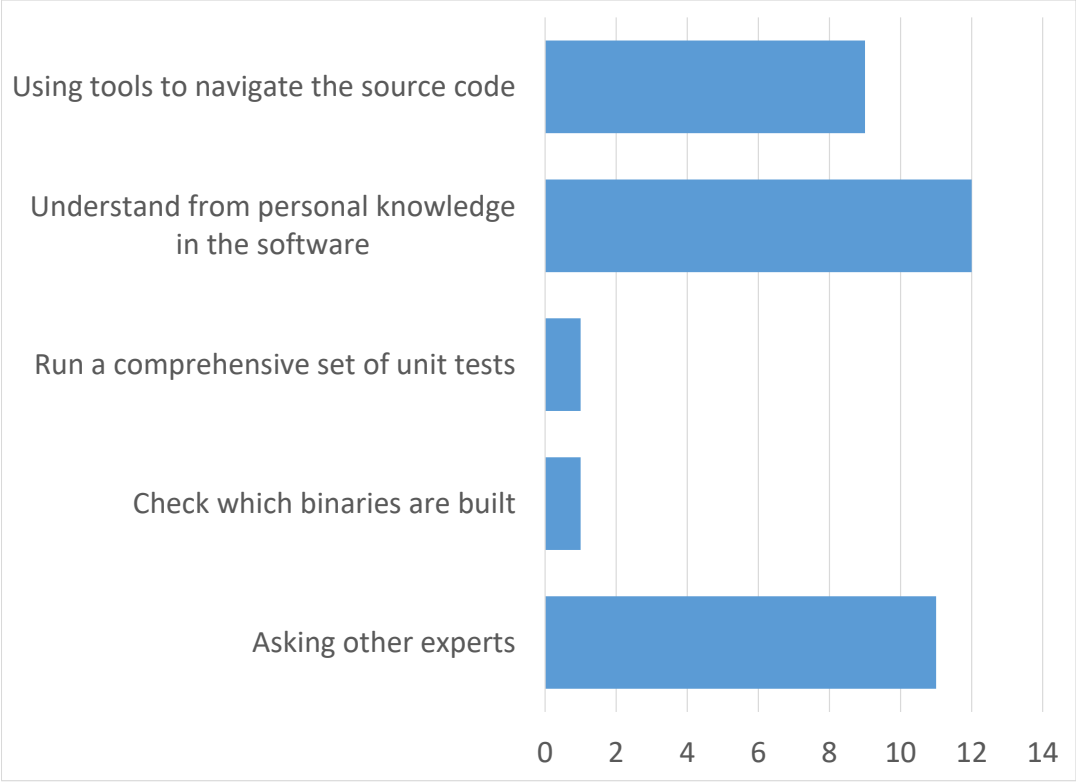


Figure 5.3: The number of developers on how they analyze the impact of a patch.

Developers on the studied Dell EMC product team are aware of the importance of impact analysis prior to code reviewing, yet few of them use dedicated impact analysis tools.

5.3.4 Change Impact Practice

Developers tend to use command line tools and their background knowledge to understand the impact of a code change. We asked developers to describe how they determine what other parts of the system is impacted by a code patch. As shown in Figure 5.3, a majority (ten of the 16 respondents) claim they use tools, such as `grep` or `find` to navigate source code to find the impact. In addition, twelve developers say that the analysis is based on their understanding from prior experience developing the software system. The respondents also described some other ways that they would use to help perform impact analysis,

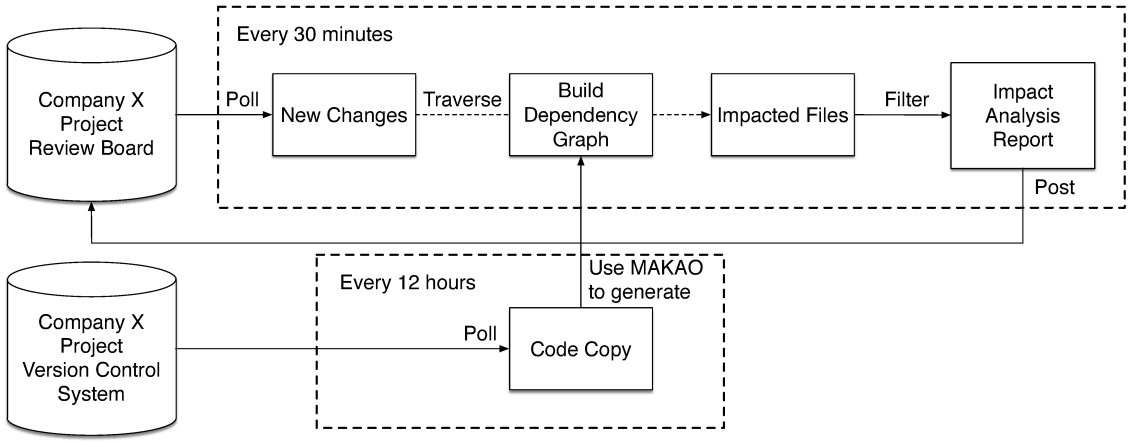


Figure 5.4: An illustration of the design of BLIMP Tracer.

including asking other experts (eleven of the 16), checking which binaries are built (one of the 16), and running unit tests (one of the 16).

Although impact analysis is one of the most crucial tasks in code review, the developers on the studied Dell EMC product team often use general-purpose command line tools to investigate the impact of patches.

5.4 BLIMP Tracer Design

To help developers conduct impact analysis accurately and efficiently, we propose BLIMP Tracer, an impact analysis tool that focuses on project deliverables. We focus on deliverable-level impact because we believe it is the most useful granularity for the review understanding task that we aim to support. In this section, we describe how BLIMP Tracer performs impact analysis for the changes posted on Dell EMC's code review platform. Figure 5.4 provides an overview of the approach, which contains changed file detection, build dependency graph extraction, graph traversal and filtering, and results presentation steps.

5.4.1 Changed File Detection

Using the Review Board API, we poll for newly posted reviews and revisions periodically. For every new change, we extract the names of the modified source code files. Those files serve as the query that we will use in later steps to trace through in order to detect the impacted deliverables.

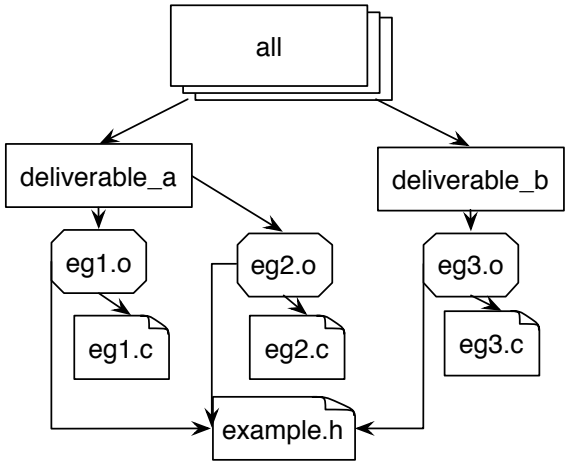


Figure 5.5: A sample build dependency graph.

Listing 5.1: A sample Makefile

```
1  CC=gcc
2  DEPS=example.h
3
4  all: deliverable_a deliverable_b
5
6  %.o: %.c $(DEPS)
7    $(CC) -c -o $@ $< -I.
8
9  deliverable_a: eg1.o eg2.o
10   $(CC) -o $@ $^
11
12 deliverable_b: eg3.o
13   $(CC) -o $@ $^
```

5.4.2 Build Dependency Graph Extraction

The build system of the studied product team is implemented using non-recursive make [50]. In make-based build systems, developers specify targets, dependencies, and rules. Targets specify an intermediate or a deliverable file. Dependencies list other targets that must exist or be updated before the target can be updated. Rules explain what command needs to be run to create the targets. For example, in Listing 5.1, line 6 specifies that all `.o` files (targets) depend on their corresponding `.c` file, as well as the `DEPS` variable that expands to the header file `example.h` (dependencies). Line 7 shows that to update the dependencies in line 6, a C compiler (in this case, `gcc`) will run to create the targets. Similar patterns can be seen in the following lines that specify rules for the deliverables.

The targets and their dependencies form what is called a Build Dependency Graph (BDG). This is a directed acyclic graph that is at the heart of the incremental build—a commodity feature of the modern build system. After executing a full build that executes all of the necessary build commands to produce project deliverables, an incremental build will only execute a subset of the full build commands that are required by activities that have taken place after the previous full build. For example, Figure 5.5 shows the BDG that corresponds to the Makefile snippet from Listing 5.1. After executing a full build, if a developer were only to update eg3.c, the build process would only re-execute the rules to eg3.o and deliverable_b.

In this chapter, we query the BDG to understand which deliverables are impacted by a set of modified files. Extraction of the BDG from a make-based build system is a non-trivial task.

We use MAKAO [1] to extract the build dependency graphs from the Dell EMC project. Since MAKAO constructs the BDG by parsing build trace logs, we fully build the project in one of the supporting Linux platforms with GNU make tracing enabled and extract the trace log.

The trace log contains a listing of commands and their corresponding files that were executed during the build job, which can be parsed by MAKAO to generate a BDG.

Extracting a BDG for the Dell EMC project requires building the software in full. At the same time, the Dell EMC project we study is large and complex, and undergoes rapid development. Indeed, conducting a full, tracing-enabled build for each uploaded review revision is impractical. Moreover, similar to our prior work [20], we find that changes that modify the BDG structure are relatively rare in practice. In order to have an up-to-date BDG for accurate impact analysis results while maintaining a low build load, BLIMP Tracer updates the BDG twice daily. We discuss the potential ramifications of this decision in Section 5.7.

5.4.3 Graph Traversal and Filtering

Once we obtain a list of files within a change, we traverse the BDG for each of the file in the list of changed files and record every target that is impacted by the change. To keep the report page readable, we filter out binary and archive files that are directly impacted by a changed file,since the impact of such files is already clear for the developers to recognize.

In addition, we identify whether the list of changed files contains build specification files (e.g. `*.mk` or `Makefile`), or newly added files that had never been shown in the repository. Since a change in the build system or adding new files may change the build dependency graph we extracted previously, a traversal on the old BDG may not provide accurate impact analysis information. In such cases, BLIMP Tracer will print a warning message in the impact analysis report page, indicating that the analysis may be incomplete. Again, in practice, BDG-changing commits are rare. Additional computing power could be used to solve this problem, but given the scarcity of the issues, the warning message was deemed sufficient for the time

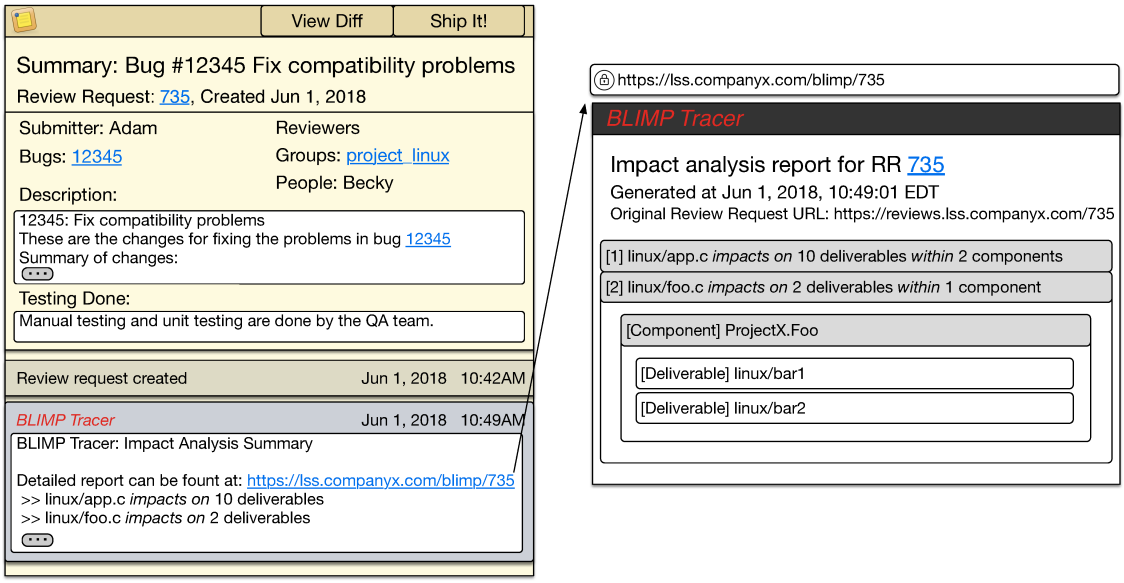


Figure 5.6: An illustration of the BLIMP Tracer interface, integrated with the code review platform. The bottom of the left image resembles a sample comment posted by BLIMP Tracer.

being.

5.4.4 Results Presentation

We generate a summary of the change impact analysis report, and post it as a code review comment to the code review platform as shown in Figure 5.6. Using an internal document that maps the (non-intermediate) build targets to the names of the deliverables of the studied project, we show the deliverables that the code patch impacts on. The names of the deliverables also correspond to internal teams that are responsible for them. Therefore, the patch authors will know who may be interested before deciding to which team this code review should be assigned.

Moreover, the summary includes how broad the code change will affect the system by displaying the number of high-level deliverables each changed file impacts. If the impact on the system is broad or affects a key customer-facing deliverable, the report can alert reviewers to more carefully assess the patch. To assist users in knowing what exactly is impacted, the summary contains a URL that points to a full impact analysis report page. The report page contains the full paths of the files and components that each changed file in the patch impacts.

Once the comment containing the summary is posted, developers and code reviewers who are desig-

Table 5.1: An overview of the interviewed developers.

Name	Dell EMC Exp. (Yrs)	Studied Proj. Exp. (Yrs)
Developer A	2	0
Developer B	13	8
Developer C	11	11
Developer D	2	2
Developer E	11	11

nated to assess the code patch will receive a notification from the code review platform, indicating that BLIMP Tracer has finished analyzing and publishing the impact analysis of the patch. Developers and reviewers can plan their code assessment activities accordingly using the information provided by BLIMP Tracer. BLIMP Tracer typically publishes reports for newly uploaded patches within 30 minutes. Note that the speed of BLIMP Tracer is not of concern because developers rarely react to a new review request more quickly than that.

5.5 User Study Design

We now discuss the design and execution of our user study with developers at Dell EMC. More specifically, we describe how we conducted semi-structured interviews and plan for future improvements.

5.5.1 An Overview of the Industrial System

BLIMP Tracer is deployed within a large, multinational product team of Dell EMC. The product suite that this team produces provides solutions for enterprise data backup and recovery. This product itself dates back to early 1990s, and has over ten million lines of code. Despite being implemented in a variety of programming languages (C, C++, Java, and C# to name a few), the product suite is highly portable, supporting product variants that run on Windows, as well as various flavours of UNIX, Linux, and macOS.

The complexity of the system and its build dependency graph make it an ideal subject system to pilot BLIMP Tracer. Indeed, understanding the large and complex build dependency graph of this product suite is difficult, even for senior Dell EMC developers.

5.5.2 Interview Design

Using the information that we gathered during the survey, we designed semi-structured interviews [65] for developers who expressed their interest in further discussing our proposed solution and how it could be improved. Semi-structured interviews contain planned open and close-ended questions, but the order is not necessarily the same as planned. In addition, during a semi-structured interview, interviewers are free to explore new findings and improvise the questions. We choose to perform semi-structured interviews instead of a more rigidly structured interview to allow for ideas that emerge during the interview to be explored to some extent. The interview sessions were recorded, transcribed, and coded.

In total, we conducted one-on-one interviews with five developers. Table 5.1 provides an overview of the participant experience levels. The participants have two to 13 years of experience working at Dell EMC (median of eleven years), and zero to eleven years of experience working on the subject product suite (median of eight years).

The purpose of the interviews was to discern whether BLIMP Tracer helps developers to perform impact analysis on patches. During the interview, we asked participants to show on screen how they assess the impacted deliverables of a code patch (without the help of BLIMP Tracer) Developers were asked to follow a think aloud protocol to enable us to gather data about why they are performing the tasks that they are performing. At the same time, we record how much time they spend. We then invite the interviewee to use BLIMP Tracer to assess the impact of the same patch, and determine if the impacted deliverables computed by BLIMP Tracer match the expected result.

5.6 User Study Results

In this section, we present the results of the semi-structured interviews with respect to effectiveness and additional benefits.

5.6.1 Effectiveness of BLIMP Tracer

One of the primary goals of BLIMP Tracer is to help developers to better understand the impact of patches. During the one-to-one interviews, we observe how developers currently conduct impact inves-

tigations for a given patch. More specifically, we selected patches that include regular `.c` files, as well as header `.h` files, to cover files that have both small and large potential impact. We invite participants to estimate the number of components and deliverables that are impacted by the patch and name some of them, using methods that are most comfortable to them. After that, we introduce BLIMP Tracer and ask developers how they would use it during code review. We record the interviews, transcribed and coded the developers response. By analyzing the transcribed and coded interview data, we identify three aspects that BLIMP Tracer can improve in the impact analysis procedures for the developers.

Using only command line tools for impact analysis is inefficient

Echoing the responses from the survey, all participants stated that when they investigate the impact of a patch for code review purposes, they usually use command line tools. However, using command line tools often does not give a full picture of the deliverables that a patch impacts. Developers B, C, and E state that if no dedicated tool for impact analysis is provided, they would only check the components that are immediately impacted by the changed file. In other words, the manual impact analysis that they conduct does not trace beyond one layer of impacted deliverables. Since these first-layer deliverables often have several transitive dependencies, the impact is likely being underestimated. Moreover, the developers agree the impact analysis process using `grep` and `find` is "*slow*". Developer A commented that for a relatively large file used by several components, "*it is impossible to do [impact analysis] by hand*".

BLIMP Tracer provides access to impact information at the right time for developers

The participants agree that integrating an impact analysis system with the code review platform is beneficial for a more well-rounded understanding of a code patch in a timely manner. Indeed, Developer C commented that although there are tools that analyze the static dependency structure of a system, he does not run the tool every time when he is asked to review a patch. Integrating a build-based impact analysis directly into the code reviewing process shows plenty promise. Indeed, Developer D explains that since doing an impact investigation of a patch takes time, he only checks the impact for the patch that he authors. The introduction of BLIMP Tracer will change the reviewing and developing behaviour. De-

veloper C stated that BLIMP Tracer would likely help him to assess "*a patch that has a lot of impacted [deliverables]*".

BLIMP Tracer improves developers' awareness of the impact that patches have on system architecture

Participants agree that integrating BLIMP Tracer with the code review platform will accelerate and improve their workflows (Developers C, D and E). Since BLIMP Tracer automatically displays impact analysis reports directly, no manual input is required for developers to see the impacted deliverables. While Developer D stated that he had used a dedicated impact assessment tool during development, the others rely on command line tools to get an approximate sense of the impact of a patch. Indeed, Developer C commented that having BLIMP Tracer in the code reviewing platform can help reviewers and patch authors to "*make sure the impacted deliverables [have been tested]*".

Developers on the studied product team at Dell EMC agree that in addition to improving development workflow, BLIMP Tracer has the potential to improve the breadth and depth of impact analysis, as well as save developer time and effort.

5.6.2 Additional Benefits of BLIMP Tracer

In addition to making impact analysis easier and faster for the developers, we wish to know how BLIMP Tracer would benefit other aspects of development. In order to do so, we asked survey participants open-ended questions after they had the opportunity to try BLIMP Tracer. More specifically, following the nature of semi-structured interview, the questions are based on the participant's comments on BLIMP Tracer during the trials. For example, if the participant made comments about the accuracy of the results of BLIMP Tracer, we would follow up with questions asking what deliverables were surprisingly included in or excluded from the impact list. The follow-up questions unveil two types of benefits that BLIMP Tracer may provide to improve understanding of the dependency structure of the studied system.

BLIMP Tracer provides knowledge to developers to reduce unnecessary dependencies in a system

First, using BLIMP Tracer, developers are able to identify deliverables that do not have a surface-visible, direct relationship with the changed files. This may expose problematic or unnecessary dependencies in the build dependency graph [13]. Removal of these unnecessary dependencies may speed up incremental builds. One theme that Developer C and E raised after examining the BLIMP Tracer report was that there were some unexpected deliverables appearing in the report. For example, Developer E commented that he found it "*odd*" that a component may be impacted by some change in a function in some other module. However, after some contemplation, he commented that it is "*understandable how the function is used in that deliverable, but I need some time to understand the logic behind it*". Knowing this dependency, he can decide whether to note this dependency, or to notify the module owners to remove the potentially unnecessary dependencies.

BLIMP Tracer helps accelerating newcomers' onboarding process

Moreover, BLIMP Tracer provides a resource that can help new developers to improve their understanding of the system architecture. A solid understanding of the project structure will reduce the risk of "shotgun surgery" [57], which would degrade the system architecture. Indeed, Developer D mentioned that when he was a newcomer to the project, the learning curve was steep to understand the connections between different project components. Therefore, he said: "*if I were a newcomer, I would use BLIMP Tracer to learn the dependency of files*".

BLIMP Tracer can help the community and developers in improving and understanding the system architecture.

5.7 Threats to Validity

In this section, we discuss the threats to the validity of our study.

5.7.1 External Validity

Threats to external validity have to do with the generalizability of our results to other subject systems. BLIMP Tracer is only integrated with the code review platform of the studied product team at Dell EMC. The focus on one project may affect the generalizability of the results. Although we have reached out to developers with various years of development and project experience, future study on other subjects may be needed to arrive at more general conclusions.

Similar to other software engineering studies, we have a low response rate with our surveys and interviews at Dell EMC. We recognize that the general population of our studied projects might have different characteristics and opinions than the ones that we present. Nevertheless, the purpose of our survey and interview is not to achieve generalizability, but rather to gather feedback and insights from practitioners who will interact with BLIMP Tracer on a daily basis.

5.7.2 Internal Validity

Threats to internal validity have to do with whether other plausible hypotheses could explain our results. We conclude that BLIMP Tracer shows promise due to our survey and interview studies. It may be that the participants were biased towards providing positive feedback to us due to social pressure. To combat this, we explained to all participants that their frank and honest feedback was what we needed to collect in order to improve BLIMP Tracer. Nevertheless, the response may still have been biased towards the positive.

Interview participants were asked to identify files that are impacted by a patch first, and then use BLIMP Tracer for confirmation in one setting. Since BLIMP Tracer was always presented second, participants may have been biased in favour of it. Although we do not suspect that the influence of tool order is strong, we plan to control for this confounding factor in future replication studies.

5.7.3 Construct Validity

Threats to construct validity have to do with the alignment of our choice of indicators with what we set out to measure. Since we generate the build impact analysis report based on the traversal of build depen-

dency graphs, changes that modify the BDG may result in inaccurate report. However, in the studied system, changes that have the potential to change the BDG are infrequent. In order to provide the most accurate BDG for generating the reports, we extract the BDG frequently (twice every day). In addition, BLIMP Tracer prints a warning message when a change to a known build specification is detected in the code patch under analysis.

BLIMP Tracer calculates the result of build impact analysis using files as a unit. Because of that, some of the 'impacted modules' in the report page may not be a direct result from the code change. Rather, the 'impacted modules' could be directly related to some other functions in the file. Our analysis is at the file-level because this is the granularity at which the Make build technology operates. Nonetheless, if a finer-grained build dependency graph were to become available, the impact analysis results would likely be even more useful for developers.

5.8 Chapter Summary

Code review is widely used in modern development process to ensure software quality. However, the mere existence of code review does not promise improvement in software quality. A key concern in code review is data-driven discussions of patch implications. To accurately assign reviewers to assess code patches and to pinpoint the potential issues that affect the system, stakeholders need to know what areas of the software will be impacted by the changes. We introduce BLIMP Tracer, a build impact analysis system that integrates with the Review Board code reviewing environment of a product team at Dell EMC. We evaluate the effectiveness of BLIMP Tracer by conducting a qualitative study. Through a study that involves semi-structured interviews with Dell EMC developers, we make the following conclusions:

- Before the introduction of BLIMP Tracer, developers often use general-purpose command line tools to analyze the build impact of a code patch.
- BLIMP Tracer not only made build impact analysis on code patches faster, but also vastly improves the depth and breath of impact analysis when compared to traditional methods.

- BLIMP Tracer can help to onboard new developers by helping them to better understand the system architecture.

6

Conclusion

Modern software development requires team collaboration. The speed and complexity at which modern software is developed make large team-based development common. At the heart of team-based development lies Modern Code Review (MCR), a mechanism that enables and enriches collaboration.

However, the mere existence of MCR does not guarantee a well-managed developer collaboration process. In fact, MCR tools lack information for stakeholders to make decisions about where to invest their time and effort. In this thesis, we empirically study how to support teams and developers in making those investment decisions. In the remainder of this section, we outline the contribution of this thesis and draw paths for future research.

6.1 Contributions and Findings

The goal of this thesis is to support project teams and developers in investing effort in code review. To do so, we analyze past data in code review and introduce an impact analysis tool that tightly integrates with

an existing code reviewing platform. We claim that:

Thesis statement: *Data about the content of past code reviews and the impact that a patch has on a software system can help stakeholders to make more effective effort-allocation decisions.*

We investigate existing code review effort allocation processes and propose tools to improve those practices. In doing so, we perform two empirical studies. Below, we reiterate the key findings of the studies presented in this thesis:

1. *Topics of Code Reviewing Feedback*

   Changes in reviewing behaviour of a code review community often coincide with project events. In addition, experienced reviewers who mature within different code reviewing communities focus on different topics according to project needs (Chapter 4).

2. *BLIMP Tracer: Build Impact Analysis for Code Review*

   Developers often use general-purpose command line tools to analyze the build impact of a code patch. BLIMP Tracer not only made build impact analysis on code patches faster, but also vastly improves the depth and breadth of impact analysis when compared to traditional methods. Moreover, BLIMP Tracer can help to onboard new developers by helping them to better understand the system architecture (Chapter 5).

6.2 Opportunities for Future Research

Although we believe that this thesis has contributed positively towards supporting the decision for investment of developer effort in code review, there are plenty of opportunities for future research. Below, we describe several paths for future work.

6.2.1 Analytics Dashboard for Tracking the Evolution of Code Reviewing Feedback

We believe that the key contribution of Chapter 4 is that it provides strong evidence of significant trends in reviewing behaviour and a set of measures that can be tracked as communities age and their stakeholders

accrue experience. To complete the last mile for decision support for developers, in future work, we would like to use our topic models to build an analytics dashboard for monitoring reviewing behaviour.

6.2.2 Filtering Code Segments from Natural Language

Although we have used a state-of-the-art machine learning library [44] for identifying documents that only contain code, the topic models we build in Chapter 4 can be improved by identifying and filtering code snippets from all text documents. Since some code tokens are also natural language elements (e.g. `ip_address`, `identifier`), removing code segments from natural language remains a complex research problem for future researchers.

6.2.3 Build Impact Analysis in the Presence of Multiple Build Tools

We acknowledge that the Dell EMC project studied in Chapter 5 is unique as it uses a single build system (`make`). Because of that, future work is needed to tackle more complex problems in the presence of multiple build tools or a fractured build graph.

6.2.4 Queryable Services for Dependency Analysis

Build dependency graph (BDG) is useful for more than supporting review effort investment decisions. Developers can use BDG to trace file dependencies in various granularity. To serve as a transition between the existing technologies that enable local dependency checks to BLIMP Tracer, it may also be helpful to offer a service to query for dependencies that offers information on the degree of dependency of an impacted deliverable.

6.2.5 Other Dimensions of Review Request Prioritization

File dependency may not be the only criterion developers use for deciding the investment of their code reviewing effort. Future data on whether the developers use other dimensions of code patches to determine the priority of review requests may also be a useful extension.

A

Additional Tables and Figures

A.1 Mapping from Topics to Terms

In this appendix section, we provide a mapping from topics to their most relevant terms for OpenStack Nova and the Dell EMC project.

Appendix A

Table A.1: A map between the OpenStack Nova topics' labels, their shares, and their top-10 key terms.

Theme	Topic Label	Share	Topic Key Terms
Context Specific	Volumes and Storage Management	6.1	instance, volume, migration, compute, call, case, state, host, delete, check
	Provisioning Decision Making	4.4	host, resource, node, scheduler, compute, instance, filter, cell, allocation, service
	Virtual Machine	5.1	libvirt, driver, image, nova, device, virt, support, type, set, default
	API Issues	10.5	api, nova, change, version, option, add, default, compute, service, note
Exception Handling	Exception Handling, Logging and User-facing Error Msg	7.3	exception, log, error, raise, message, check, case, return, code, debug
Language Specific	Python Collections	8.1	object, instance, field, list, return, key, uuid, dict, set, method
Design	Object Oriented Design	12.4	method, make, code, function, call, class, thing, change, bit, object
Code Review Process	Code Review Process and Minor Issues	20	comment, patch, change, good, inline, nit, commit, bug, message, code
Code Style	Code Style	6.8	line, nit, import, nova, space, file, remove, blank, pep, python
Unit Testing	Unit Testing	6.2	test, unit, mock, method, case, add, call, testing, called, fake

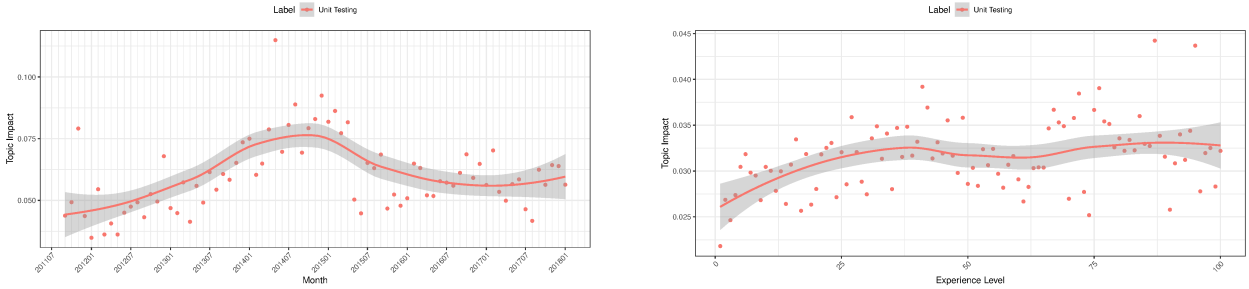
Appendix A

Table A.2: A map between the Dell EMC project topics' labels, their shares, and their top-10 key terms.

Theme	Topic Label	Share	Topic Key Terms
Context Specific	File Locations	4.5	file, nsr, directory, cpp, change, path, lib, build, version, library
	Project Configuration	6.5	device, change, user, snapshot, data, backup, restore, password, set, group
	Project Terminology	8.8	backup, set, save, client, time, server, change, database, job, list
Exception Handling	Logging and User-Facing Error Msg	9.3	message, error, log, user, change, unable, file, debug, add, suggest
	Exception Handling and Memory Management	8.8	null, check, return, error, free, line, set, false, true, call
Language Specific	String/Buffer Issues	5.4	string, const, char, std, size, buffer, return, int, type, function
Design	Object Oriented Design and Concurrency	10.4	code, function, class, make, method, call, case, time, thread, file
	Function Design (low-level)	11.9	function, comment, line, code, add, remove, variable, file, move, call
Code Review Process	Code Review Process and Minor Issues	14.8	change, comment, code, review, good, issue, test, check, fix, add
Code Style	Code Style	8.2	line, space, remove, add, change, delete, file, tab, quote, indentation

A.2 Additional Figures

In this appendix section, we provide additional figures (Figure A.1) for the *unit testing* topic in OpenStack Nova. Because we neither observe trend of *unit testing* topic in Nova, nor a topic that share a similar theme in the Dell EMC project, we did not include the figures in the observation.



(a) Topic trend over time.

(b) Topic trend over experience.

Figure A.1: The figures of OpenStack Nova's *unit testing* topic trends.

References

[1] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 114–123. IEEE, 2007.

[2] Amritanshu Agrawal, Wei Fu, and Tim Menzies. What is wrong with topic modeling? and how to fix it using search-based software engineering. *Information and Software Technology*, 2018.

[3] Jafar M Al-Kofahi, Hung Viet Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Detecting semantic changes in makefile build code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 150–159. IEEE, 2012.

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 432–441. ACM, 2005.

[5] Robert S Arnold and Shawn A Bohner. Impact analysis-towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, pages 292–301. IEEE, 1993.

[6] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[7] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 375–384. ACM, 2010.

[8] Vipin Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, volume 2, pages 931–940, 2013.

[9] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.

[10] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. The influence of non-technical factors on code review. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 122–131. IEEE, 2013.

[11] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 202–211. ACM, 2014.

[12] Nicolas Bettenburg, Ahmed E Hassan, Bram Adams, and Daniel M German. Management of community contributions. *Empirical Software Engineering*, 20(1):252–289, 2015.

[13] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering*, 22(6):3117–3148, 2017.

[14] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 26–26. IEEE, 2007.

[15] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022, 2003.

[16] Amiangshu Bosu and Jeffrey C Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 33. ACM, 2014.

[17] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 146–156. IEEE, 2015.

[18] Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 55–65. IEEE, 2006.

[19] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 105–111. ACM, 2006.

[20] Qi Cao, Ruiyin Wen, and Shane McIntosh. Forecasting the duration of incremental build jobs. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 524–528. IEEE, 2017.

[21] Jonathan Chang, Jordan L Boyd-Graber, Sean Gerrish, Chong Wang, and David M Blei. Reading tea leaves: How humans interpret topic models. In *Nips*, volume 31, pages 1–9, 2009.

[22] Tse-Hsun Chen, Stephen W Thomas, and Ahmed E Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

[23] Jonathan Corbet and Greg Kroah-Hartman. *Linux Kernel Development Report*. The Linux Foundation, 2017.

[24] David Roxbee Cox and Alan Stuart. Some quick sign tests for trend in location and dispersion. *Biometrika*, 42(1/2):80–95, 1955.

[25] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: how the current code review best practice slows us down. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 27–28. IEEE Press, 2015.

[26] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. A security perspective on code review: The case of chromium. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30. IEEE, 2016.

[27] Michael E Fagan. Design and code inspections to reduce errors in program development. In *Pioneers and Their Contributions to Software Engineering*, pages 301–334. Springer, 2001.

[28] Daniel M German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. Was my contribution fairly reviewed? a framework to study fairness in modern code reviews. In *Proceedings of the International Conference on Software Engineering (ICSE)*, page To appear. ACM, 2018.

[29] Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, Stefano Maffulli, and Gregorio Robles. Understanding How Companies Interact with Free Software Communities. *IEEE Software*, 30(5):38–45, 2013.

[30] Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, Gregorio Robles, and Alvaro del Castillo. Analyzing Gerrit Code Review Parameters with Bicho. In *Proceedings of the International Workshop on Software Quality and Maintainability (SQM)*, pages 1–12, 2014.

[31] Alex Gyori, Shuvendu K Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 318–328. ACM, 2017.

[32] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, Ana Erika Camargo Cruz, Kenji Fujiwara, and Hajima Iida. Who Does What during a Code Review? Datasets of OSS Peer Review Repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 49–52, 2013.

[33] Abram Hindle, Christian Bird, Thomas Zimmermann, and Nachiappan Nagappan. Do topics make sense to managers and developers? *Empirical Software Engineering*, 20(2):479–515, 2015.

[34] Abram Hindle, Michael W Godfrey, and Richard C Holt. What's hot and what's not: Windowed developer topic analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 339–348. IEEE, 2009.

[35] Mohammad-Amin Jashki, Reza Zafarani, and Ebrahi Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 84–90. ACM, 2008.

[36] Yujuan Jiang, Bram Adams, and Daniel M German. Will my patch make it? and how fast? case study on the linux kernel. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 101–110. IEEE, 2013.

[37] Dan Jurafsky. *Speech & language processing*. Pearson Education, 2000.

[38] Nafiseh Kahani, Mojtaba Bagherzadeh, Juergen Dingel, and James R Cordy. The problems with eclipse modeling tools: a topic analysis of eclipse forums. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 227–237. ACM, 2016.

[39] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1028–1038. ACM, 2016.

[40] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120. IEEE, 2015.

[41] Victor Lavrenko and W Bruce Croft. Relevance based language models. In *Proceedings of the International Conference on Research and Development in Information Retrieval*, pages 120–127. ACM, 2001.

[42] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[43] Erik Linstead, Cristina Lopes, and Pierre Baldi. An application of latent dirichlet allocation to analyzing software evolution. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, pages 813–818. IEEE, 2008.

[44] Mika V Mäntylä, Fabio Calefato, and Maelick Claes. Natural Language or Not (NLoN) - A Package for Software Engineering Text Analysis Pipeline. In *Proc. of the 15th International Conf. on Mining Software Repositories (MSR)*, page to appear, 2018.

[45] Mika V Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *Transactions on Software Engineering (TSE)*, 35(3):430–448, 2009.

[46] Girish Maskeri, Santonu Sarkar, and Kenneth Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the India Software Engineering Conference*, pages 113–120. ACM, 2008.

[47] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[48] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 192–201. ACM, 2014.

[49] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[50] Peter Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc*, 19(1):14–25, 1998.

[51] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 503–512. ACM, 2002.

[52] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 171–180. IEEE, 2015.

[53] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit Software Code Review Data from Android. In *Proc. of the 10th Working Conf. on Mining Software Repositories (MSR)*, pages 45–48, 2013.

[54] Takuto Norikane, Akinori Ihara, and Kenichi Matsumoto. Which review feedback did long-term contributors get on oss projects? In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 571–572. IEEE, 2017.

[55] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. Are developers aware of the architectural impact of their changes? In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 95–105. IEEE Press, 2017.

[56] Thai Pangsakulyanont, Patanamon Thongtanunam, Daniel Port, and Hajimu Iida. Assessing MCR discussion usefulness using semantic similarity. In *Proceedings of the International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 49–54. IEEE, 2014.

[57] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[58] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[59] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the International Conference on Software Engineering (ICSE) Companion*, pages 222–231. ACM, 2016.

[60] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[61] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, pages 202–212. ACM, 2013.

[62] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.

[63] Peter C Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 541–550. ACM, 2011.

[64] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.

[65] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, 2009.

[66] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the International Conference on Software Engineering*, pages 724–734. ACM, 2014.

[67] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, David Pursehouse, and Naoyasu Ubayashi. Why are Commits being Reverted? A Comparative Study of Industrial and Open Source Projects. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301––311, 2016.

[68] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 1–5. ACM, 2005.

[69] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 650–660. IEEE Press, 2012.

[70] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 99–108. IEEE Press, 2015.

[71] Yee Whye Teh, Michael I Jordan, Matthew J Beal, and David M Blei. Sharing clusters among related groups: Hierarchical dirichlet processes. In *Nips*, pages 1385–1392, 2004.

[72] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. Improving Code Review Effectiveness through Reviewer Recommendations. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 119–122, 2014.

[73] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1039–1050, 2016.

[74] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review Participation in Modern Code Review: An Empirical Study of the Android, Qt, and OpenStack Projects. *Empirical Software Engineering*, 22(2):768–817, 2017.

[75] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[76] Patanamon Thongtanunam, Xin Yang, Norihiro Yoshida, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. ReDA: A Web-based Visualization Tool for Analyzing Modern Code Review Dataset. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 605–608, 2014.

[77] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in github. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 144–154. ACM, 2014.

[78] Qiang Tu and Michael W Godfrey. An integrated approach for studying architectural evolution. In *Proceedings of the International Workshop on Program Comprehension*, pages 127–136. IEEE, 2002.

[79] Perry van Wesel, Bin Lin, Gregorio Robles, and Alexander Serebrenik. Reviewing Career Paths of the OpenStack Developers. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 543–548, 2017.

[80] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, page To appear, 2018.

[81] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who Should Review This Change? Putting Text and File Location Analyses Together for More Accurate Recommendations. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–270, 2015.

[82] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Tag recommendation in software information sites. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 287–296. IEEE, 2013.

[83] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. Mining the modern code review repositories: A dataset of people, process and product. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 460–463. IEEE, 2016.

[84] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically Recommending Peer Reviewers in Modern Code Review. *Transactions on Software Engineering (TSE)*, 42(6):530–543, 2016.

[85] Weizhong Zhao, James J Chen, Roger Perkins, Zhichao Liu, Weigong Ge, Yijun Ding, and Wen Zou. A heuristic approach to determine an appropriate number of topics in topic modeling. *BMC Bioinformatics*, 16(13):S8, 2015.

[86] Yu Zhao, Feng Zhang, Emad Shihab, Ying Zou, and Ahmed E Hassan. How are discussions associated with bug reworking?: An empirical study on open source projects. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 21:1–21:10. ACM, 2016.

[87] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: from patch-based to pull-request-based tools. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 871–882. ACM, 2016.